O'REILLY®

图灵程序设计丛书

第2版

SVG精髓

SVG Essentials



[美] J. David Eisenberg
[加] Amelia Bellamy-Royds 著
易郑超 何鹏飞 译

中国工信出版集团 人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

译者介绍

易郑超

网名TooBug。Web前端工程师。现就职于富途网络，曾在腾讯CDC任职前端工程师。关注Web前端领域前沿技术。

何鹏飞

网名basecss，就职于腾讯CDC。喜欢前端技术，好阅读。

TURING 图灵程序设计丛书

SVG精髓（第2版）

SVG Essentials

Second Edition

［美］J. David Eisenberg ［加］Amelia Bellamy-Royds 著
易郑超　何鹏飞 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人 民 邮 电 出 版 社
北　　京

图书在版编目（ＣＩＰ）数据

SVG精髓 : 第2版 / (美) 艾森伯格
(Eisenberg,J.D.) , (加) 贝拉米-罗伊斯
(Bellamy-Royds,A.) 著 ; 易郑超, 何鹏飞译. -- 北京 :
人民邮电出版社, 2015.10
（图灵程序设计丛书）
ISBN 978-7-115-40254-7

Ⅰ. ①S… Ⅱ. ①艾… ②贝… ③易… ④何… Ⅲ. ①
图形软件 Ⅳ. ①TP391.41

中国版本图书馆CIP数据核字(2015)第201196号

内 容 提 要

本书通过实例透彻讲解了SVG（可缩放矢量图形）这种标记语言的规范及应用。作者从简单的SVG应用开始，带领读者逐步探索了SVG的复杂功能，包括滤镜、变换、渐变和模式。从应用层面看，本书涵盖了动画、交互图形和动态SVG编程等技术，不仅能为有经验的开发人员提供重要参考，同时通过讲解基本的XML和CSS技术，为没有Web开发经验的读者提供了入门捷径。

本书适合Web及移动Web开发人员阅读参考。

◆ 著　[美] J. David Eisenberg
　　[加] Amelia Bellamy-Royds
译　易郑超　何鹏飞
责任编辑　岳新欣
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　　　　印刷

◆ 开本：800×1000　1/16
印张：18.5
字数：437千字　　2015年10月第1版
印数：1－3 500册　　2015年10月北京第1次印刷
著作权合同登记号　图字：01-2015-4679号

定价：69.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

版权声明

© 2015 by J. David Eisenberg and Amelia Bellamy-Royds.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2015. Authorized translation of the English edition, 2014 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由 O'Reilly Media, Inc. 出版，2014。

简体中文版由人民邮电出版社出版，2015。英文原版的翻译得到 O'Reilly Media, Inc. 的授权。此简体中文版的出版和销售得到出版权和销售权的所有者——O'Reilly Media, Inc. 的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

献给我已故的母亲和父亲，感谢他们多年来给我的爱和建议。

——JDE

献给比尔，他会为我感到骄傲的。

——ABR

目录

前言

本书将向你介绍“可缩放矢量图形”（Scalable Vector Graphics）技术，即 SVG。SVG 是万维网联盟（W3C）的一项推荐标准，它使用 XML 来描述由直线、曲线、文本等组成的图形。这段干巴巴的定义并不能体现出 SVG 的作用和它的强大之处。

你可以将 SVG 图形加到 XSL-FO（Extensible Stylesheet Language Formatting Objects）[1] 文档中，然后将文档转换为 Adobe PDF 格式来获得更高的印刷质量。地图和气象领域的工作者可以使用 SVG 来创建高精度、高质量、可移植的图形。Web 开发者将 SVG 嵌入网页来创建高分辨率的响应式图形，且可以使文件尺寸很小。本书中的所有图表最初都是由 SVG 创建的。在学习和使用 SVG 时，你一定能想到这项新技术的一些新的、有趣的使用场景。

本书读者

如果你想做以下事情，就应该读一读这本书：

- 在文本编辑器或者 XML 编辑器中创建 SVG 文件
- 从已有的矢量数据创建 SVG 文件
- 将其他 XML 数据转换为 SVG
- 使用 JavaScript 操作 SVG 文档对象树

选错书的读者

如果你只是想查看 SVG 文件，只需要安装一个阅读器或者 Web 插件，然后下载 SVG 文件查看就可以了。这种情况下你并不需要知道背后的原理，除非你想满足自己强烈的好奇心。

如果你想使用带有 SVG 导出功能的图像处理软件来创建 SVG 文件，那么只需要阅读相关

注 1：一种用于文档格式的 XML 标记语言，可参见 http://zh.wikipedia.org/wiki/XSL-FO。——译者注

软件的文档来学习如何使用软件的功能就可以了。

如果你打算继续阅读……

如果你确实适合阅读这本书，那么你应该了解，本书的大部分读者都是高级用户，他们很可能有技术背景，而不是图形设计背景。所以我们不打算在前面讲很多非常基础的东西，但我们希望没有 XML 或者程序设计背景的人也能阅读本书，因此也准备了一些介绍性的章节，并将它们放到本书最后的附录中。如果你没有使用过 XML 或者样式表（这可能包括一些技术人员），也没有编写过程序，可能需要先翻到附录部分。稍后，我们会概述各章和附录的主要内容。

如果你是技术工作者，也需要知道，本书并不能将你变成一位艺术家，就像一本讲字处理算法的书并不能让你把文章写得更好一样。本书将展示 SVG 的很多技术细节，而如果要成为艺术家，你还需要学习观察。除了本书之外，你还应该读读 Betty Edwards 博士的 *The New Drawing on the Right Side of the Brain*[2]。

本书只会给出 SVG 的一些基本信息，如果你想了解所有信息，请参考万维网联盟的 SVG 规范（http://www.w3.org/Graphics/SVG/Overview.htm8）。

关于示例

本书中的所有示例，除了涉及 HTML 页面的之外，全部在运行在 GNU/Linux 系统上的 Batik SVG viewer 软件中测试通过。Batik SVG viewer 是由 Apache 软件基金会下的 Batik 项目开发的一款软件。这款软件使用 Java 开发，跨平台，并遵循 Apache 软件协议开源，可以从 http://xmlgraphics.apache.org/batik 下载。

书中的所有例子（包括第 2、13 和 14 章中涉及 JavaScript 和 HTML 的例子）通过在 Firefox 和 Chrome 浏览器中加载的方式进行了测试。对 SVG 高级特性的支持程度取决于浏览器。

你在看本书中的示例的时候，会发现它们完全没有任何艺术价值。这是有原因的。首先，每个示例都是为了展示 SVG 的一个方面，那么它就应该只展示这一个方面，而不应该有其他的视觉干扰。其次，本书作者 David 在看其他书中那些漂亮得不可思议的图形时感到很沮丧，他心想："我永远也画不出这么漂亮的图。"为了不让你产生同样的沮丧情绪，我们有意简化了这些示例。当你看到它们的时候，你的第一反应会是："我可以用 SVG 画出比这漂亮得多的东西！"你当然可以，然后你就会动手去画。

注 2：该书中文版《五天学会绘画》已由北方文艺出版社出版。http://book.douban.com/subject/5263615/。
——译者注

本书结构

- 第 1 章　入门指南

  本章简要介绍了 SVG 的历史，比较了栅格图形系统与矢量图形系统，最后用一个简单的教程介绍了 SVG 的主要概念。

- 第 2 章　在网页中使用 SVG

  本章展示了在 HTML5 文档中使用 SVG 的各种方法。

- 第 3 章　坐标系统

  如何在画图时确定一个点的位置？哪个方向是“上”？本章解答了这些问题，并展示了如何切换图形中的坐标系统。

- 第 4 章　基本形状

  本章展示了如何使用 SVG 中的基本形状来构成一个图形，这些基本形状有：线、长方形、多边形、圆、椭圆。本章也讨论了如何指定形状的轮廓和内部颜色。

- 第 5 章　文档结构

  在复杂的图形中，会有一些元素被复用或者是重复出现。本章会教你如何将对象组合成一个整体，使它们变成一个实体，可以复用。本章也讨论了如何使用外部的矢量图形或者栅格图形。

- 第 6 章　坐标系统变换

  如果你在一种可伸缩的材料上画一个正方形，然后水平拉伸材料，就会得到一个长方形。将材料的两个对边分别往不同的方向斜切，你会得到一个平行四边形。再将这个材料旋转 45 度，你会得到一个菱形。在本章中，你将学到如何对坐标系统进行移动、旋转、缩放、斜切，以改变画布上图形的形状。

- 第 7 章　路径

  所有的基本形状都是“路径”这个一般概念的特殊实例。本章将展示如何使用线、圆弧和复杂的曲线来描述形状的一般轮廓。

- 第 8 章　图案和渐变

  本章在第 4 章的基础上增加了关于颜色的讨论，比如如何创建渐变色或者如何创建填充模式。

- 第 9 章　文本

  一张图并不只有线和形状，文本也是海报或者示意图的重要组成部分。本章展示了如何添加文字，包括沿直线分布的文字和沿指定路径分布的文字。

- 第 10 章　裁剪和蒙版

  本章展示了如何使用裁剪路径（clipping path），让图形变得好像是从圆形镜头、锁孔或者其他形状中观察到的一样。本章还会展示如何使用遮罩来改变对象的透明度，使得对象的边缘呈现出“淡出”效果。

- 第 11 章　滤镜

  尽管 SVG 文件是用来描述矢量图形的，但文档最终还是在栅格设备上被渲染的。在本章中，你会学到如何应用面向栅格图形的滤镜，来使图形变得模糊，改变它的颜色，或者产生一些灯光效果。

- 第 12 章　SVG 动画

  本章展示了如何使用 SVG 内置的动画能力。

- 第 13 章　添加交互

  除了使用 SVG 内置的动画，你还可以使用 CSS 和 JavaScript 来动态控制图形的属性。

- 第 14 章　使用 SVG DOM

  本章进一步深入讨论了使用 JavaScript 操作文档对象模型。还简要介绍了为使用 SVG 而设计的一个 JavaScript 库。

- 第 15 章　生成 SVG

  尽管可以从零开始创建一个 SVG 文件，但很多人希望将已有的矢量数据或者 XML 数据以图表的形式展现出来。本章将讨论使用编程语言和 XSLT 来从已有数据创建 SVG 的方法。

- 附录 A　SVG 中需要的 XML 知识

  SVG 是 XML（Extensible Markup Language，可扩展标记语言）的一种应用。如果你没有使用过 XML，应该读一下本附录，以熟悉这项用来组织数据和文档的异常强大和灵活的技术。

- 附录 B　样式表介绍

  你可以使用样式表来为 SVG 文档中的特定元素指定一些视觉属性。它们和 HTML 文档使用的样式表几乎完全一样。如果你没有使用过样式表，应该看看这部分对样式表的简要介绍和解析。

- 附录 C　编程概念

  如果你是图形设计工作者，没有太多编程经验，应该读一下这部分，以便了解程序员说的“对象模型”和“函数”是什么。

- 附录 D　矩阵代数

  尽管不是很有必要，但是要完全理解 SVG 中的坐标变换和滤镜效果的话，理解矩阵代

数是很有帮助的。矩阵代数就是用来计算坐标和像素的数学知识。本附录重点关注矩阵代数的基础部分。

- 附录 E　创建字体

  TrueType 字体会以矢量图形（字符）来展示文字。本附录展示了如何使用你最喜欢的字体并将它们转换为 SVG 文档中可以使用的路径。

- 附录 F　将圆弧转换为不同的格式

  很多软件中的圆弧使用的是“中心 + 角度”的格式。本附录提供了将这种格式与 SVG 格式互相转换的代码。

排版约定

本书使用了下列排版约定。

- 楷体

  表示新术语或强调的内容。

- 等宽字体（`Constant width`）

  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`Constant width bold`**）

  表示应该由用户输入的命令或其他文本。



该图标表示提示、建议或一般注记。



该图标表示警告或警示。

本书使用编号来表示代码清单中有趣的点。编号采用实心圆加数字的形式。代码清单的下方给出了相应的解释。这里有一个例子：

```
Roses are red,
    Voilets are blue. ❶
Some poems rhyme;
    This one doesn't. ❷
```

❶ 紫罗兰的实际颜色值为 #9933cc。

❷ 这首诗使用的文学手法叫作意外的结局。

许多示例都可以在线测试，并且文中都给出了 URL。有些在线示例还包含可以编辑的标记；点击刷新按钮可以查看改变后的结果。也可以点击重置按钮回到示例的原始状态。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920032335.do

对于本书的评论和技术性问题，请发送电子邮件到：bookquestions@oreilly.com。

要了解更多 O’Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：
http://www.oreilly.com

我们在 Facebook 的地址如下：http://facebook.com/oreilly

请关注我们的 Twitter 动态：http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：http://www.youtube.com/oreillymedia

第1版致谢

感谢本书编辑 Simon St. Laurent，他总是能给出非常中肯的指导和意见。他还在电子邮件中告诉我“我们相信你知道该如何写”，这是我听到的最美妙的话语之一。

感谢 Edd Dumbill，我略微地改动了一下他编写的文档，就形成了附录 A。当然，该附录中的任何错误都是因为我的修改而造成的。

感谢本书的技术审阅人：Antoine Quint、David Klaphaak 以及 Adobe 的 SVG 质量工程团队，他们对初稿做了技术审阅。他们的意见从许多方面提升了本书的质量。

Jeffrey Zeldman 是第一个让我脑海中产生写书想法的人，真诚地感谢他。

感谢所有人，尤其是我的哥哥 Steven。当我告诉他我正在写一本书时，他十分信任我，并对我说：“哇，太棒了！”

第2版致谢

感谢 Shelly Powers 进行了出色的技术审阅。还要感谢 Simon St. Laurent 和 Meghan Blanchette 出色的编辑工作。尽管挑剔的作者尽了最大的努力，但还是由 Matthew Hacker 和 O’Reilly 的工具和产品团队做了一切收尾工作，感谢他们。

David 的致谢：特别感谢 Amelia Bellamy-Royds。她一开始对本书进行了技术审阅，她的点评清晰易懂，写得非常好，以至于我一字不差地照搬了，所以她应该是一位合著者。她的修正和补充让本书变得更好，远远超出我的想象。

Amelia 的致谢：感谢 David 对我所做的额外的工作给予了充分的认可和赞誉。本书第 1 版就是非常受欢迎的介绍 SVG 的图书。作为一个曾经对浏览器实现的各种怪异行为感到困惑不解的过来人，我真的希望这一版能将我在学习 SVG 时的困惑一一解释清楚。

我还要特别感谢我的丈夫 Chris，他给了我极大的支持，但也时常提醒我暂停工作，去吃饭、睡觉或者去呼吸些新鲜空气。

第 1 章

入门指南

SVG，即可缩放矢量图形（Scalable Vector Graphics），是一种 XML 应用，可以以一种简洁、可移植的形式表示图形信息。目前，人们对 SVG 越来越感兴趣。大多数现代浏览器都能显示 SVG 图形，并且大多数矢量绘图软件都能导出 SVG 图形。本章首先介绍两大计算机图形系统，然后讨论 SVG 的适用情境，最后分析一个简单的示例，其中用到的很多概念将在后续章节中详细探讨。

1.1　图形系统

计算机中描述图形信息的两大系统是栅格图形（raster graphics）和矢量图形（vector graphics）。

1.1.1　栅格图形

在栅格图形系统中，图像被表示为图片元素或者像素的长方形数组（如图 1-1 所示）。每个像素用其 RGB 颜色值或者颜色表内的索引表示。这一系列像素也称为位图（bitmap），通常以某种压缩格式存储。由于大多数现代显示设备也是栅格设备，显示图像时仅需要一个阅读器将位图解压缩并将它传输到屏幕上。

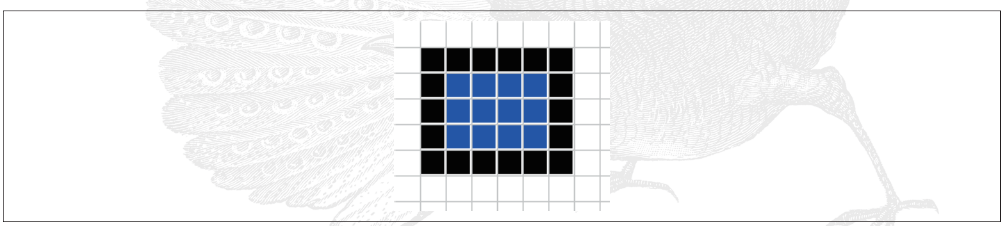


图 1-1：栅格长方形

1.1.2　矢量图形

在矢量图形系统中，图像被描述为一系列几何形状（如图 1-2 所示）。矢量图形阅读器接受在指定坐标集上绘制形状的指令，而不是接受一系列已经计算好的像素。

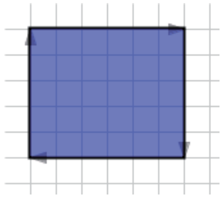


图 1-2：矢量长方形

想象一下在一张绘图纸上作图的过程，栅格图形的工作就像是描述哪个方格应该填充什么颜色，而矢量图形的工作则像是描述要绘制从某个点到另一个点的直线或曲线。有些人把矢量图形描述为一组绘图指令，而位图（栅格图形）则是在特定的位置填充颜色的点。矢量图形“知道”它们是什么——方块“知道”它是一个方块，文本“知道”它是文本。由于矢量图形是对象而不是一系列像素，因此矢量对象可以改变它们的形状和颜色，而位图则不能。此外，所有文本都是可搜索的，因为无论看起来是什么样子或者做了怎样的旋转或变换，它们实际上还是文本。

还可以将栅格图形想象为画布上的绘画，而矢量图形则是由可伸缩材料构成的直线和形状，它们可以在背景上移动。

1.1.3　栅格图形的用途

栅格图像最适合用来表示照片，因为照片很少由明显的线条和曲线组成。扫描的图像也通常被存储为位图，即使它最初可能是一张线图，但人们也希望存储的是整个图像，而并不关心它的各个组成部分。比如传真机就不关心你绘制的是什么，它只是使用栅格图形将像素从一个地方传输到另一个地方。

创建栅格格式图像的工具很多，而且通常比许多矢量图形的工具更好用。压缩和存储栅格图像的方式（格式）有很多种，并且这些格式的内部规则都是公开的。用于读写 JPEG、GIF 和 PNG 等压缩格式的程序库唾手可得。这也是 SVG 出现之前 Web 浏览器只支持栅格图像的部分原因。

1.1.4　矢量图形的用途

矢量图形用于以下领域。

- 计算机辅助绘图（Computer Assisted Drafting，CAD）程序，因为精确地测量和放大绘图以便查看细节非常重要。
- 设计用于高分辨率打印图像的程序，例如 Adobe Illustrator。
- Adobe PostScript 打印和成像语言，打印的每个字符都用直线和曲线来描述。
- 基于矢量图形的 Macromedia[1] Flash 系统，用来设计动画、演示和网站。

由于大多数这类文件都编码为二进制格式或打包好的比特流，所以有一些工作很难做，比如让浏览器或者其他用户代理 2 解析内嵌的文本，或者让服务器基于外部数据动态创建矢量图形文件。大多数矢量图形的内部规则都是专用的，浏览、创建这些图形的代码很难获得。

1.2 可缩放

尽管矢量图形不像栅格图形那么流行，但它可以缩放而不损失图像质量，因而在许多应用程序中具有不可估量的价值。例如，这里有两个猫图像。图 1-3 是栅格图像，图 1-4 是一个矢量图像。它们都显示在一个 PPI[3] 为 72 的屏幕上。

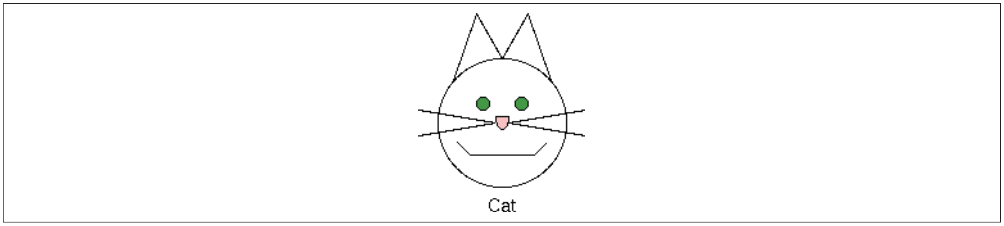


图 1-3：栅格形式的猫图像

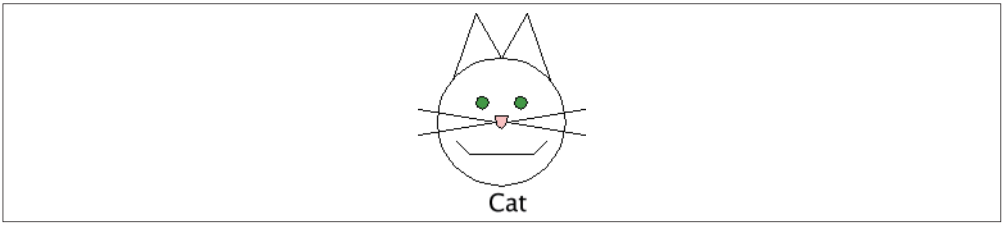


图 1-4：矢量形式的猫图像

当显示程序放大栅格图像时，它必须以某种方式扩大每个像素。要想将图像放大至四倍，最简单的方式就是让每个像素放大为原来的四倍。如图 1-5 所示，结果并不是很理想。

注 1：Macromedia 已被 Adobe 收购。旗下产品也分别改名。——译者注
注 2：指浏览网页或者其他文件的设备或软件。——译者注
注 3：PPI，即 pixels per inch，每英寸的像素点数量。 ——译者注

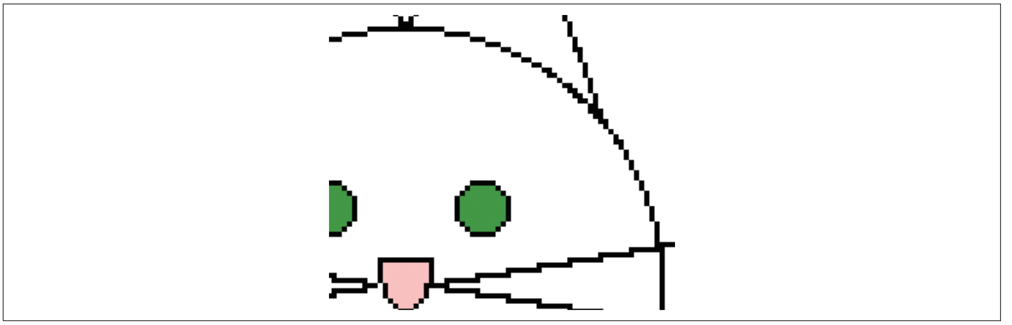


图 1-5：放大的栅格图像

尽管可以使用边缘检测和反锯齿这类技术优化放大后的图像，但是这些技术很耗时。此外，由于栅格图像中的所有像素都是未知的，因此并不能保证相关算法能正确检测到边缘的形状。反锯齿的结果看起来如图 1-6 所示。

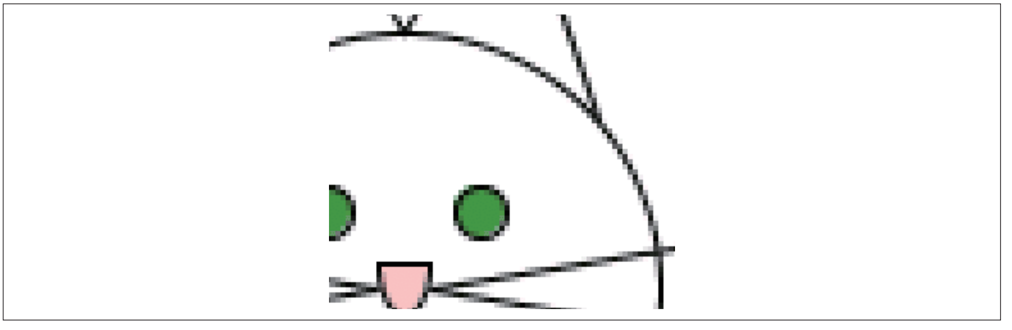


图 1-6：放大后的反锯齿栅格图像

另一方面，将矢量图像放大为原来的四倍时，只需图像显示程序将形状的所有坐标都乘以 4，然后用显示设备的完整分辨率重新绘制它们即可。因此，在如图 1-7 所示的 DPI 为 72 的屏幕截图中，线条边缘很清晰，与放大后的栅格图像相比，锯齿明显少多了。

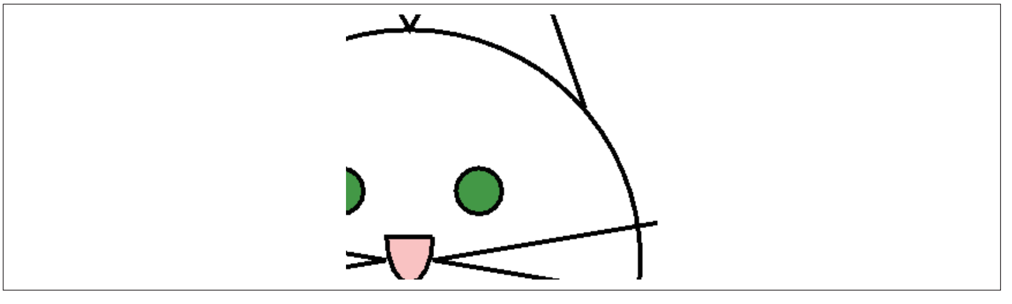


图 1-7：放大后的矢量图形

1.3 SVG的作用

1998 年，万维网联盟成立了一个工作小组，负责开发作为 XML 应用的矢量图形表示方法。由于 SVG 是 XML 程序，所以图像的有关信息被存储为纯文本，而且它还具有 XML 的开放性、可移植性以及可交互性。

CAD 和图形设计程序通常使用特定的二进制格式存储绘图信息。当它们拥有导入和导出 SVG 格式图像的能力后，这类应用程序就有了一个通用的标准格式来交换信息。

由于 SVG 就是一个 XML 应用，因此它能与其他 XML 处理程序结合使用。例如，数学教科书可以使用 XSL 来对说明性文本进行格式化对象，使用 MathML 描述方程，以及使用 SVG 为方程生成图表。

SVG 工作组制订的规范是一个万维网联盟官方推荐规范。诸如 Adobe Illustrator 和 Inkscape 这类应用程序都可以导入和导出 SVG 格式的绘图。在 Web 中，许多浏览器都原生支持 SVG，而且 SVG 具有很多与 HTML 中 CSS 样式相同的变换和动画能力。由于 SVG 文件就是 XML，因此其中的文本可以被使用任何能够解析 XML 的用户代理读取显示。

1.4 创建一个SVG图像

本节你会看到一个 SVG 文本，内容是本章前面见过的猫的图像。这个例子介绍了很多概念，在后续章节中会对这些概念进行详细介绍。这个文件很好地演示了如何编写示例文件，但你在创建项目中的 SVG 文件时，并不一定要按照这种方式编写。

1.4.1 文档结构

示例 1-1 以标准的 XML 处理指令和 `DOCTYPE` 声明开始。根元素 `<svg>` 以像素为单位定义了整个图像的 `width` 和 `height`，还通过 `xmlns` 属性定义了 SVG 的命名空间。`<title>` 元素的内容可以被阅读器显示在标题栏上或者是作为鼠标指针指向图像时的提示，`<desc>` 元素允许我们为图像定义完整的描述信息。

示例 1-1：SVG 文档的基本结构 [4]

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="140" height="170" xmlns="http://www.w3.org/2000/svg">
<title>Cat</title>
<desc>Stick Figure of a Cat</desc>
<!-- 在这里绘制图像 -->
</svg>
```

注 4：当前稳定的 XML 和 SVG 版本都是 1.1。更多信息可参考：W3C-SVG1.1 规范（http://www.w3.org/TR/SVG11/）和 W3C-XML 规范（http://www.w3.org/TR/2006/REC-xml11-20060816）。——译者注

1.4.2　基本形状

你可以添加一个 <circle> 元素来绘制猫的脸部。这个元素的属性指定中心点 x 坐标和 y 坐标以及半径。点 (0,0) 为图像的左上角。水平向右移动时 x 坐标增大，垂直向下移动时 y 坐标增大。

这个圆的位置和尺寸是绘图结构的一部分。绘图的颜色是表现（presentation）的一部分。按照 XML 程序的惯例，为了保持最大的灵活性，应该分离结构和表现。表现信息包含在 `style` 属性中。它的值是一系列表现属性和值，正如附录 B 中所述。这里轮廓的画笔颜色为黑色，填充颜色为 `none` 以使猫的脸部透明。SVG 见示例 1-2，其结果见图 1-8。

示例 1-2：基本形状——圆

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-02.html

```
<?xml version="1.0"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg">
<title>Cat</title>
<desc>Stick Figure of a Cat</desc>

<circle cx="70" cy="95" r="50" style="stroke: black; fill: none"/>

</svg>
```

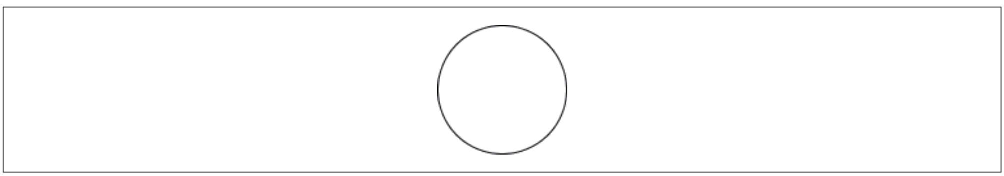


图 1-8：阶段一，绘制圆

1.4.3　指定样式属性

接下来我们在示例 1-3 中添加两个圆作为眼睛。虽然填充颜色和画笔颜色也是表现的一部分，但是 SVG 允许我们使用单独的属性指定它们。在这个示例中，填充（`fill`）和轮廓画笔颜色（`stroke`）写在两个单独的属性中，而不是全部写在 `style` 属性中。你可能并不会经常使用这种方法，在 5.2.4 节会详细讲述这些内容。这里提及它只是为了证明可以这么做。结果如图 1-9 所示。

为了节省空间，这里省略了 `<?xml ...?>` 和 `<!DOCTYPE?>`。

示例 1-3：基本形状——填充圆

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-03.html

```
<svg width="140" height="170" xmlns="http://www.w3.org/2000/svg">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none"/>
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
</svg>
```



图 1-9：阶段二，绘制脸和眼睛

1.4.4　图形对象分组

示例 1-4 使用两个 `<line>` 元素在猫的右脸上添加了胡须。我们想把这些胡须作为一个部件（稍后你会明白为什么），所以把它们包装在分组元素 `<g>` 里面，然后给它一个 `id`。我们可以通过指定起点和终点 *x* 坐标和 *y* 坐标（分别为 `x1` 和 `y1` 以及 `x2` 和 `y2`）的方式绘制一条直线。结果如图 1-10。

示例 1-4：基本形状——线

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-04.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg">
<title>Cat</title>
<desc>Stick Figure of a Cat</desc>

<circle cx="70" cy="95" r="50" style="stroke: black; fill: none;" />
<circle cx="55" cy="80" r="5" stroke="black" fill="#339933" />
<circle cx="85" cy="80" r="5" stroke="black" fill="#339933" />
<g id="whiskers">
   <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;" />
   <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;" />
</g>
</svg>
```

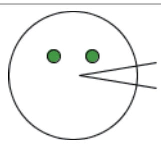


图 1-10：阶段三，在右脸上添加胡须

1.4.5　变换坐标系统

现在我们使用 <use> 复用胡须分组并将它变换（transform）为左侧胡须。示例 1-5 中，首先在 scale 变换中对 x 坐标乘以 −1，翻转了坐标系统。这意味着原始坐标系统中的点 (75,95) 现在位于 (−75,95)。在新的坐标系统中，向左移动会使坐标增大。这就意味着必须将坐标系统向右 translate（平移）140 个像素（负值），才能将它们移到目标位置，如图 1-11 所示。

示例 1-5：变换坐标系统

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-05.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;" />
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933" />
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933" />
  <g id="whiskers">
    <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;" />
    <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;" />
  </g>
  <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)" />
</svg>
```

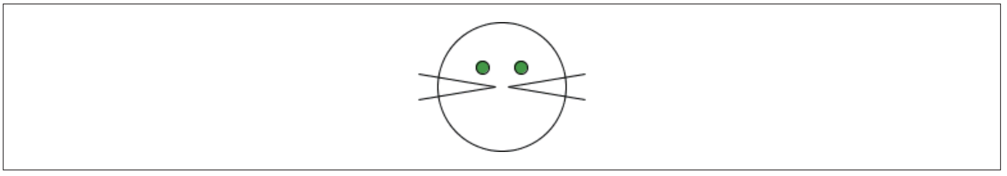


图 1-11：阶段四，添加左侧胡须

<use> 元素中的 xlink:href 属性在不同的命名空间中（详情请查看附录 A）。为了确保 SVG 文档能在所有 SVG 阅读器中工作，我们必须在开始的 <svg> 标签中添加 xmlns:xlink 属性。

transform 属性依次列出了所有的变换，不同的变换之间使用空格分隔。

1.4.6　其他基本图形

示例 1-6 使用 <polyline> 元素构建了嘴和耳朵，它接受一对 x 和 y 坐标作为 points 属性的值。你可以根据喜好使用空格或者逗号分隔这些数值。结果如图 1-12 所示。

示例 1-6：基本图形——折线

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-06.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;" />
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933" />
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933" />
  <g id="whiskers">
    <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;" />
    <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;" />
  </g>
  <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)" />
  <!-- 耳朵 -->
  <polyline points="108 62, 90 10, 70 45, 50, 10, 32, 62"
    style="stroke: black; fill: none;" />
  <!-- 嘴 -->
  <polyline points="35 110, 45 120, 95 120, 105, 110"
      style="stroke: black; fill: none;" />
</svg>
```

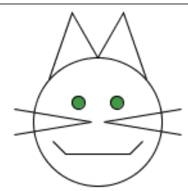


图 1-12：阶段五，添加耳朵和嘴

1.4.7　路径

所有的基本形状实际上都是通用的 <path> 元素的快捷写法，示例 1-7 中使用 <path> 元素为猫添加了鼻子，结果如图 1-13 所示。这个元素被设计用来以尽可能简洁的方式指定路径或者一系列直线和曲线。示例 1-7 中的路径翻译过来就是“移动到坐标 (75,90)。绘制一条到坐标 (65,90) 的直线。然后以 *x* 半径为 5、*y* 半径为 10 绘制一个椭圆，最后回到坐标 (75,90) 处”。

示例 1-7：使用 <path> 元素

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-07.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
```

```
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
  <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
  <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
  <g id="whiskers">
    <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;"/>
    <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;"/>
  </g>
  <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)"/>
  <!-- 耳朵 -->
  <polyline points="108 62, 90 10, 70 45, 50, 10, 32, 62"
    style="stroke: black; fill: none;" />
  <!-- 嘴 -->
  <polyline points="35 110, 45 120, 95 120, 105, 110"
      style="stroke: black; fill: none;" />
  <!-- 鼻子 -->
  <path d="M 75 90 L 65 90 A 5 10 0 0 0 75 90"
    style="stroke: black; fill: #ffcccc"/>
</svg>
```

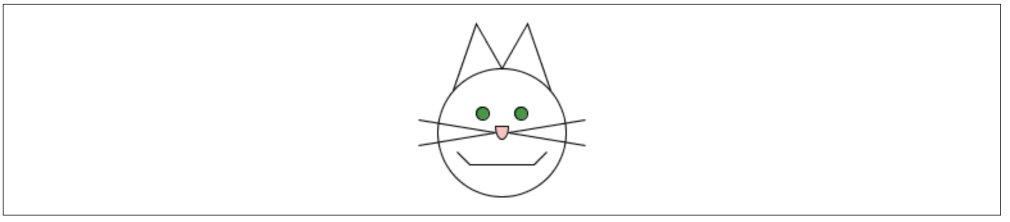


图 1-13：阶段六，添加鼻子

1.4.8 文本

最后，由于这个图像绘制得很粗糙，所以用户很可能看不出这是一只猫。因此，示例 1-8 为这个图像添加了一些文本作为标记。在 `<text>` 元素中，`x` 和 `y` 属性用于指定文本的位置，它们也是结构的一部分。字体和字号是表现的一部分，因而也是 `style` 属性的一部分。与你见过的其他元素不同，`<text>` 是一个容器元素，它的内容是你想要显示的文本。图 1-14 展示了最终结果。

示例 1-8：添加标记

http://oreillymedia.github.io/svg-essentials-examples/ch01/ex01-08.html

```
<svg width="140" height="170"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
  <title>Cat</title>
  <desc>Stick Figure of a Cat</desc>

  <circle cx="70" cy="95" r="50" style="stroke: black; fill: none;"/>
```

```
    <circle cx="55" cy="80" r="5" stroke="black" fill="#339933"/>
    <circle cx="85" cy="80" r="5" stroke="black" fill="#339933"/>
    <g id="whiskers">
      <line x1="75" y1="95" x2="135" y2="85" style="stroke: black;"/>
      <line x1="75" y1="95" x2="135" y2="105" style="stroke: black;"/>
    </g>
    <use xlink:href="#whiskers" transform="scale(-1 1) translate(-140 0)"/>
    <!-- 耳朵 -->
    <polyline points="108 62, 90 10, 70 45, 50, 10, 32, 62"
      style="stroke: black; fill: none;" />
    <!-- 嘴 -->
    <polyline points="35 110, 45 120, 95 120, 105, 110"
        style="stroke: black; fill: none;" />
    <!-- 鼻子 -->
    <path d="M 75 90 L 65 90 A 5 10 0 0 0 75 90"
      style="stroke: black; fill: #ffcccc"/>
    <text x="60" y="165" style="font-family: sans-serif; font-size: 14pt;
      stroke: none; fill: black;">Cat</text>
  </svg>
```

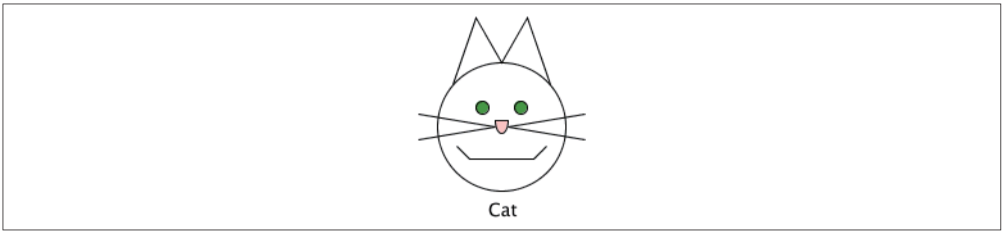


图 1-14：阶段七，带有文本标记的完整图像

有关 SVG 的简单介绍到此结束，接下来的几章将会深入探究这些概念。

第2章

在网页中使用SVG

约翰 · 邓恩说过，没有人是孤岛，同样 SVG 也不是孤立存在的。当然，你可以将 SVG 图像看作是在 Web 浏览器或者 SVG 阅读器中的一个独立文件。本书中的许多示例都是这样运作的。但是在另外一些情况下，我们希望将图形集成在一个较大的文档中，其中包含文本、表单或者其他仅使用 SVG 无法轻松显示的内容。本章描述了将 SVG 集成到 HTML 以及其他类型文档中的各种方式。

图 2-1 展示了第 1 章中绘制的猫，分别以 4 种不同的方式插入 HTML 页面。结果看起来几乎相同，但是每种方法都有其优点和局限性。

2.1 将SVG作为图像

SVG 是一种图像格式，因此可以使用与其他图像类型相同的方式包含在 HTML 页面中。具体可以采用两种方法：将图像包含在 HTML 标记的 `<img>` 元素内（当图像是页面的基本组成部分时，推荐这种方式）；或者将图像作为另一个元素的 CSS 样式属性插入（当图像主要用来装饰时，推荐这种方式）。

将 SVG 文件作为图像包含进来时，无论使用哪种方法，都具有一定的局限性。图像渲染（即“绘制”，也就是 SVG 代码被转换为栅格图像以用于显示）时与主页面是分离的，而且无法在两者之间进行通信。主页面上的样式对 SVG 无效。所以如果你的图像包含文本或者要定义相对于字体大小的长度值，那么你可能需要在 SVG 代码内部定义一个默认字体大小。此外，运行在主页面上的脚本也无法感知或者修改 SVG 的文档结构。

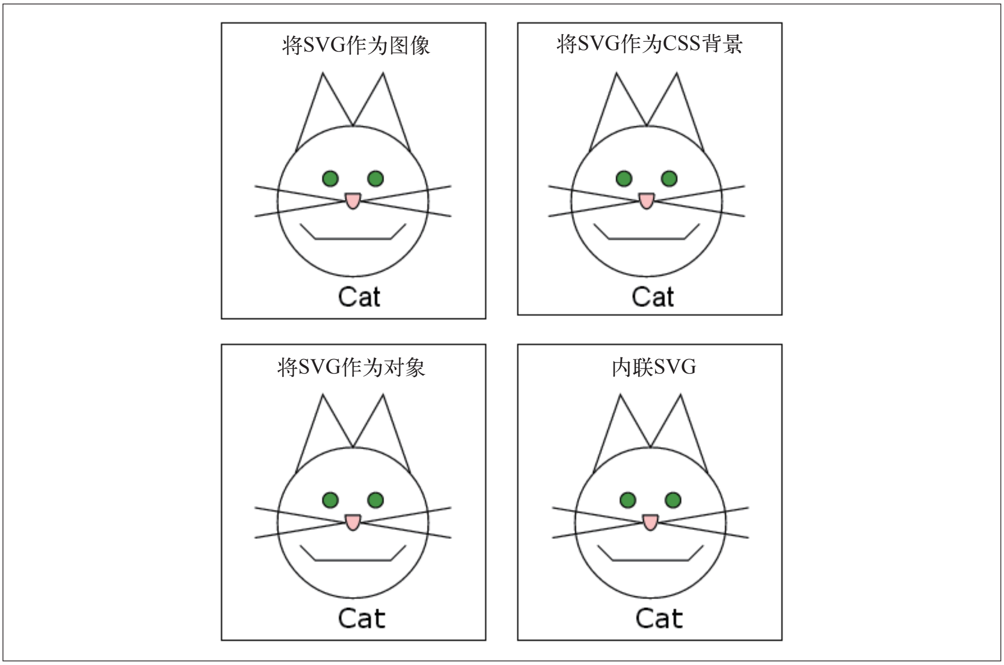


图 2-1：使用四种方式在 Web 页面中插入 SVG 的屏幕截图

在 SVG 被作为图像引用时，大多数 Web 浏览器都不会加载 SVG 自己引用的文件，包括其他图像文件、外部脚本，甚至是 Web 字体文件。根据浏览器以及用户的安全设置不同，SVG 文件内定义的脚本也可能不会运行，URL 片段（URL 中 # 后面的部分，表示文件的哪部分是你感兴趣的）也可能被忽略。在支持 SVG 图像的浏览器中，图像中的动画（详见第 12 章）是支持的。

2.1.1 在<img>元素内包含SVG

HTML `<img>` 元素定义了一个空间，表示浏览器应该将图像绘制到这个空间中。要使用的图像文件指定在 `src`（source）属性内。在 `<img>` 元素内包含 SVG 图像非常简单，只需设置 `src` 指向 Web 服务器上 SVG 文件的位置即可。当然，你还应该使用 `alt` 或者 `title` 属性给出图像描述文本，以便用户在不能看到图像时仍然能理解它代表什么。例如：

```
<img src="cat.svg" title="Cat Image"
     alt="Stick Figure of a Cat" />
```

> 虽然大多数 Web 浏览器现在都支持使用 SVG 作为图像，但一些老式浏览器却不知道该如何渲染文件，会显示一个破碎的文件图标（或者什么都不显示）。还有一些浏览器可能需要你确认你的 Web 服务器为以 `.svg` 结尾的文件声明了正确的媒体类型头（`image/svg+xml`）。

图像的高度和宽度可以使用属性或者 CSS 属性（优先考虑）设置。其他 CSS 属性控制 Web 页面内图像的位置。如果不指定 `<img>` 元素的尺寸，就会使用图像文件固有的尺寸。如果只指定高度（宽度），宽度（高度）就会按比例缩放，以使高宽比（宽高比）与图像文件固有的尺寸匹配。

对于栅格图像来说，它固有的尺寸就是它的像素尺寸。对于 SVG 来说，则更为复杂。如果文件中的根元素 `<svg>` 带有明确的 `height` 和 `width` 属性，则它们会被用作文件的固有尺寸。如果只指定 `height` 或者 `width` 而不是两个都指定，并且 `<svg>` 带有 `viewBox` 属性，那么将用 `viewBox` 计算宽高比，图像也会被缩放以匹配指定的尺寸。如果 `<svg>` 带有 `viewBox` 属性而没有尺寸，则 `viewBox` 的 `height` 和 `width` 将被视为像素长度。如果这些听起来都难以理解，别担心，我们会在 3.3 节适当地介绍 `viewBox` 属性。

如果 `<img>` 元素和 `<svg>` 根元素都没有任何有关图像尺寸的信息，浏览器应该为嵌入内容应用默认 HTML 尺寸，通常是 150 像素高、300 像素宽，但是最好不要依赖默认尺寸。

2.1.2 在CSS中包含SVG

许多 CSS 样式属性都接受一个指向图像文件的 URL 作为属性值。最常用的便是 `background-image` 属性，它会在应用样式的元素的文本内容后面绘制这个图像（或者多个叠加的图像）。

默认情况下，背景图像会按照固有尺寸进行绘制，并且会在垂直和水平两个方向上重复，以填满该元素。SVG 文件的固有尺寸用 2.1.1 节描述的方式确定。如果没有固有尺寸，SVG 会被缩放为元素高度和宽度的 100%。这个尺寸可以使用 `background-size` 属性显示地设置，重复模式和图像位置可以使用 `background-repeat` 和 `background-position` 设置：

```
div.background-cat {
 background-image: url("cat.svg");
 background-size: 100% 100%;
}
```

当多个小图标和标识使用栅格图像时，通常会将所有图像放在一个图像文件的网格内，然后使用 `background-size` 和 `background-position` 为每个元素设置对应的图像。这样，浏览器只需下载一个图像文件，从而可使网页显示得更快。这种组合图像文件被称为 CSS 精灵，这一命名寓意着神奇的精灵让事情变得更简单。随着浏览器的渲染性能越来越高，SVG 文件也可以被设计为精灵，但是我们应该尽量避免使精灵文件太大。

SVG 规范还定义了在一个图像文件中创建多个图标的其他方式，然后我们可以使用 URL 片段指示要显示哪个图标。理想情况下，这将取代基于 `background-position` 属性的精灵。然而，正如前面提到的，当将 SVG 作为图像渲染时有些浏览器会忽略 URL 片段，因此这些特性目前在 CSS 中并没有太大的用途。

除了用作背景图像，在 CSS 中 SVG 文件还能用作 `list-image`（用于创建装饰性项目列表）或者 `border-image`（用于创建花哨的边框）。

2.2　将SVG作为应用程序

要将外部 SVG 文件整合到 HTML 页面中，而又不想受到作为图像嵌入时的种种限制的话，可以使用嵌入对象。

`<object>` 元素是嵌入外部文件到 HTML（第 4 版及以上版本）以及 XHTML 文档中一种通用方式。它可以用于嵌入图像，类似于 `<img>`，也可以用于嵌入独离的 HTML/XML 文档，类似于 `<iframe>`。更重要的是，它还可以用于嵌入任意类型的文件，只要浏览器有解析该文件类型的应用程序（浏览器插件或者扩展）即可。使用 `<object>` 嵌入 SVG，可以让那些不能直接显示 SVG 但是有 SVG 插件的老版本浏览器用户也能查看图像。

`<object>` 元素的 `type` 属性表示要嵌入的文件类型。这个属性应该是一个有效的网络媒体类型（通常被称为 MIME 类型）。对于 SVG，使用 `type="image/svg+xml"`。

浏览器使用文件类型确定如何（或者是否可以）显示该文件，而不需要首先下载文件。文件的位置通过 `data` 属性指定。`alt` 和 `title` 属性的用法和图像一样。

`object` 元素必须有起始标签和结束标签。这两个标签之间的内容只会在对象数据本身不能被渲染时显示。这可以用来指定在浏览器无法显示 SVG 时应显示的备用图像或者警告文本。[1] 下面的代码会在浏览器不支持 SVG 时显示一个说明文本和一个栅格图像：

```
<object data="cat.svg" type="image/svg+xml"
    title="Cat Object" alt="Stick Figure of a Cat">
  <!-- 文本或者栅格图像用作备用选项 -->
  <p>No SVG support! Here's a substitute:</p>
  <img src="cat.png" title="Cat Fallback"
    alt="A raster rendering of a Stick Figure of a Cat"/>
</object>
```

<object> 与 <embed>

引入 <object> 元素之前，有些浏览器使用非标准的 <embed> 元素来达到同样的目的。现在 <embed> 已经被标准采用了，因此如果需要支持老版本浏览器，可以使用 <embed> 元素替代 <object> 元素。为了更广泛的支持，可将 <embed> 作为 <object> 标签内部的备用内容。

<embed> 和 <object> 之间有两个重要的区别：首先，<embed> 中源数据文件使用 `src` 而不是 `data` 属性指定；其次，<embed> 元素不能包含任何子内容，因此如果嵌入失败就没有备用选项。

虽然规范没有采用，但是在大多数浏览器中 <embed> 元素还支持可选的 `pluginspage` 属性，用于指定一个插件安装页面的地址，供之前没有安装过的用户下载安装渲染插件。

注 1：除了备用内容，元素可能还包含定义插件参数的元素。但是它们并不用于 SVG 渲染。

当 SVG 文件作为嵌入对象引入时（无论是使用 `<object>` 还是 `<embed>`），SVG 文件的渲染方式与它被包含在 `<img>` 元素中时大致相同：它会被缩放以适配嵌入元素的宽高，并且不会继承定义在父文档中的任何样式。

然而，与图像不同的是，嵌入的 SVG 可以包含外部文件，同时脚本可以在该对象和父页面之间进行通信，正如 13.4.5 节所介绍的那样。

2.3　混合文档中的SVG标记

把 SVG 作为图像和应用程序嵌入 Web 页面中是显示一个单独、完整的 SVG 文件的两种方法。然而，我们也可以在一个文件同时包含 SVG 代码与 HTML 或者 XML 标记。

将标记合并到一个文件中可以缩短 Web 页面的加载时间，因为浏览器无需单独下载图像文件。但是，如果同一图像用于站点中的多个页面，则在每个页面中重复出现的 SVG 标记会增加总体积和下载时间。

更重要的是，当应用 CSS 样式和使用脚本时，混合文档内的所有元素会被视为一个文档对象。

2.3.1　SVG中的foreign object

混合内容的一种方式是在 SVG 内插入部分 HTML（或其他）内容。SVG 规范定义了一种在图像指定区域嵌入这种“foreign”内容的方式。

`<foreignObject>` 元素定义了一个矩形区域，Web 浏览器（或者其他 SVG 阅读器）应该在其中绘制子 XML 内容。浏览器负责确定如何绘制内容。子内容通常是 XHTML（XML 兼容的 HTML）代码，但它也可能是 SVG 阅读器能显示的任意形式的 XML。内容类型通过子内容上的 `xmlns` 属性声明的 XML 命名空间来定义。

矩形绘制区域通过 `<foreignObject>` 元素的 `x`、`y`、`width` 和 `height` 属性定义，方式类似于 `<use>` 或者 `<image>` 元素，我们将在第 5 章中详细讲述。

矩形区域基于本地 SVG 坐标系统求值，因此受坐标系统变换（将在第 6 章讨论）和其他 SVG 效果影响。子 XML 文档通常渲染在矩形框内，其结果可以像其他 SVG 图像一样被操作。一个包含 XHTML 段落的 SVG foreign object 效果如图 2-2 所示。

对于创建混合 SVG/XHTML 文档，`<foreignObject>` 元素极具潜力，但是目前未得到很好的支持。IE（到版本 11 为止）根本不支持它，在其他浏览器实现中也还存在错误和不一致性。

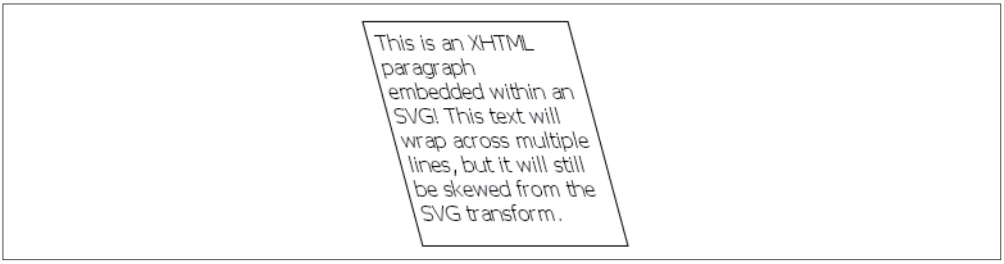


图 2-2：包含 XHTML 文本的 SVG 文件截图

如果想定义备用内容，以防 SVG 阅读器不能显示 foreign object 内容，可以结合使用 `requiredFeatures` 属性和 `<switch>` 元素，如示例 2-1 所示。在支持 XHTML 和 foreign object 的浏览器中，这段代码的效果如图 2-2 所示；在其他浏览器中，它只显示 SVG 文本。

示例 2-1：包含 `<switch>` 元素的 `<foreignObject>` 元素

```
<g transform="skewX(20)">
<switch>
  <!-- 选择一个子元素 -->
  <foreignObject x="1em" y="25%" width="10em" height="50%"
     requiredFeatures="http://www.w3.org/TR/SVG11/feature#Extensibility">
     <body xmlns="http://www.w3.org/1999/xhtml">
        <p>This is an XHTML pragraph embedded within a SVG!
           So this text will wrap nicely around multiple lines,
           but it will still be skewed from the SVG transform.
        </p>
     </body>
  </foreignObject>
  <text x="1em" y="25%" dy="1em">
    This SVG text won't wrap, so it will get cut off...
  </text>

</switch>
</g>
```

`<switch>` 元素会告诉 SVG 阅读器只有在元素的 `requiredFeatures`、`requiredExtensions` 以及 `systemLanguage` 属性求值为 `true` 或者缺失时，才绘制第一个直接子元素（以及所有子节点）。9.7.2 节会讨论使用 `systemLanguage` 属性在不同文本之间切换。测试需要使用的特性时，使用规范（http://www.w3.org/TR/SVG11/feature）提供的 URL 字符串之一即可；支持 `<foreignObject>` 是可扩展特性的一部分。

> 不幸的是，并没有一种一致的、跨浏览器的方式来指定哪种类型的 foreign object 是必需的。你可能想用 MathML 语言来为图表展示公式，然后为不认识 MathML 的浏览器指定一个纯文本版本的备用。`requiredExtensions` 属性可以指示需要使用哪种类型的 foreigh object，但是 SVG1.1 规范并没有明确地描述应该如何指定扩展——只是说它应该是一个 URL。FireFox 使用 XML 命名空间 URL，但是其他浏览器不是。

2.3.2 在XHTML或者HTML5中内联SVG

另一种混合 SVG 和 XHTML 的方式是在 XHTML 文档中包含 SVG 标记，它还可以在使用 HTML5 语法的非 XML 兼容的 HTML 文档中使用。这种在 Web 页面中包含 SVG 的方式称为内联 SVG（Inline SVG），以区别于将 SVG 作为图像或者对象嵌入的方式，尽管它实际上应该被称为 Infile SVG，因为并没有要求 SVG 代码必须出现在一行中。

2012 年及之后发布的所有主流桌面 Web 浏览器以及大多数最新的移动浏览器都支持内联 SVG。对于 XHTML，可以通过在 SVG 命名空间内定义 SVG 元素来表明正在切换到 SVG。要做到这一点，最简单的方式就是在顶级 `<svg>` 元素上设置 `xmlns="http://www.w3.org/2000/svg"`，它会改变该元素以及其所有子节点的默认命名空间。对于 HTML5 文档（使用 `<!DOCTYPE html>` 的文件），在标记中可以跳过命名空间声明。HTML 解析器会自动辨别 `<svg>` 元素和它的子节点都在 SVG 命名空间内，除了 `<foreignObject>` 元素的子元素。

在 (X)HTML 文档中插入 SVG 标记比在 SVG 标记中插入 (X)HTML 文档容易，因为无需单独的类似于 `<foreignObject>` 的元素定义在哪里渲染 SVG。相反，可以给 `<svg>` 元素自身应用定位样式，让它成为图形的框架。

默认情况下，定位 SVG 时采用内联显示模式（这意味着它和前后的文本会被插入到同一行），并且其尺寸会基于 `<svg>` 元素的 `height` 和 `width` 属性决定。使用 CSS 时可以通过设置 `height` 和 `width` CSS 属性改变尺寸，使用 `display`、`margin`、`padding` 和许多其他 CSS 定位属性改变其定位。[2]

示例 2-2 给出了一个非常简单的 HTML5 文档中的非常简单的 SVG 绘图的代码。其结果如图 2-3 所示。对于 HTML5，`<svg>` 元素上的 `xmlns` 属性是可选的。对于 XHTML 文档，需要改变文件顶部的 `DOCTYPE` 声明，以及使用 `<![CDATA[...]]>` 块包裹 `<style>` 元素内的 CSS 代码。

示例 2-2：HTML 文件中的内联 SVG

```
<!DOCTYPE html>
<html>
<head>
  <title>SVG in HTML</title>
  <style>

svg {
    display: block; ❶
    width: 500px;
    height: 500px;
    margin: auto;
    border: thick double navy; ❷
    background-color: lightblue;
}
```

注 2：CSS 定位属性应用于顶级元素，也就是 HTML 元素的直接子元素。为另一个 SVG 元素的子元素时，它会基于嵌套 SVG 规则进行定位，正如第 3 章所述。

```
body {
  font-family: cursive; ❸
}
circle {
  fill: lavender; ❹
  stroke: navy;
  strike-width: 5;
}

  </style>
</head>
<body>
  <h1>Inline SVG in HTML Demo Page</h1>
    <svg viewBox="0 0 250 250"
       xmlns="http://www.w3.org/2000/svg">
    <title>An SVG circle</title>
    <circle cx="125" cy="125" r="100"/>
    <text x="125" y="125" dy="0.5em" text-anchor="middle">
       Look Ma, Same Font!</text>
  </svg>
  <p>And here is regular HTML again...</p>
</body>
</html>
```

❶ 第一个样式规则定义了 SVG 应该如何放置以及在 HTML 文档内的尺寸。

❷ 也可以使用其他 CSS 属性为要绘制的 SVG 盒子指定样式。

❸ 定义给主文档的样式会被 SVG 继承。

❹ 也可以在主样式表内为 SVG 元素定义样式。

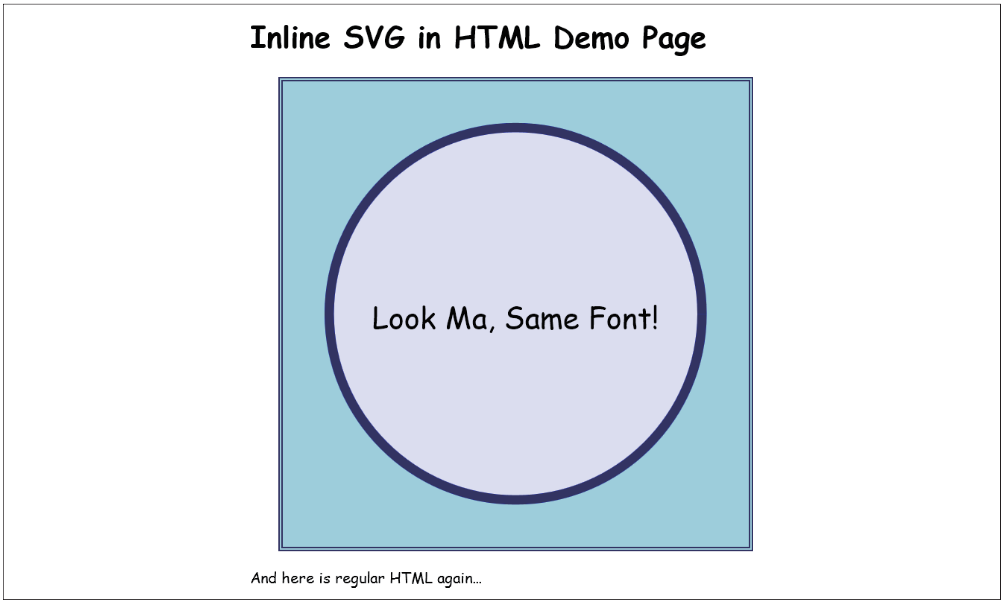


图 2-3：示例 2-2 的结果

2.3.3 其他XML应用程序中的SVG

XML 命名空间除了可以用于 XHTML 以外，也可用于在其他 XML 文档中标识 SVG。具体的细节依赖于主 XML 文档的语法，但是有两个基本要求：XML 文档必须为 SVG 元素明确定义一个布局盒子，并且用于显示这个文档的程序必须知道如何绘制 SVG。

一种经常内联 SVG 的 XML 文档类型是可扩展样式表语言格式化对象（Extensible Stylesheet Language Formatting Object，XSL-FO）文件。一个 XSL-FO 文件定义了一个多页文档的内容和布局，可以用于发布或者创建一个 PDF 文件。XSL-FO 数据类型定义包含一个 `<instream-foreign-object>` 元素，它类似于 SVG 的 `<foreignObject>` 元素，定义一个矩形区域来容纳来自不同命名空间的内容。我们可以在里面添加 SVG 标记。只需确保 `<svg>` 标签以及其所有子元素都定义在 SVG 命名空间内，为此需要为所有 SVG 元素启用命名空间前缀，或者使用 `xmlns` 属性改变默认命名空间。

示例 2-3 给出了一个 XSL-FO 文件的代码片段，为格式化对象元素使用了自定义的 `fo` 命名空间前缀。SVG 命名空间被设置为 `<svg>` 及其子元素的默认命名空间，因此在图像标记内无需使用前缀。

示例 2-3：XSL-FO 文档内的 SVG

```
<?xml version="1.0" encoding="UTF-8"?>
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <!-- 其他格式化对象内容 -->
  <fo:instream-foreign-object width="140px" height="140px">
    <svg xmlns="http://www.w3.org/2000/svg"
         width="140px" height="140px">
      <!-- 在此输入SVG代码 -->
    </svg>
  </fo:instream-foreign-object>
  <!-- 文档的其他内容 -->
</fo:root>
```

第3章

坐标系统

SVG 的世界就是一张无限大的画布。在本章中，我们将了解到如何使阅读器知道在这张画布中应该显示哪一部分区域，如果确定它的尺寸，以及如何在该区域内确定点的位置。

3.1　视口

文档打算使用的画布区域称作*视口*。我们可以在 `<svg>` 元素上使用 `width` 和 `height` 属性确定视口的大小。属性的值可以是一个数字，该数字会被当作用户坐标下的像素。也可以指定 `width` 和 `height` 为带有单位的数字，单位的取值是下列值之一。

- `em`
  默认字体的大小，通常相当于文本行高。

- `ex`
  字母 *x* 的高度。

- `px`
  像素（在支持 CSS2 的图形系统中，每英寸为 96 像素）。

- `pt`
  点（1/72 英寸）。

- `pc`
  12 点（1/6 英寸）。

- `cm`

  厘米。

- `mm`

  毫米。

- `in`

  英寸。

以下是几种合法的 SVG 视口声明形式：

```
<svg width="200" height="150">
<svg width="200px" height="150px">
```

这两个声明都指定了一个 200 像素宽、150 像素高的区域。

```
<svg width="2cm" height="3cm">
```

这个声明指定了一个 2 厘米宽、3 厘米高的区域。

```
<svg width="2cm" height="36pt">
```

混用单位是可行的，但是并不常用。这个元素指定了一个 2 厘米宽、36 点高的区域。

还可以指定 `<svg>` 元素的 `width` 和 `height` 为百分比。当元素嵌套在另一个 `<svg>` 元素里时，其百分比根据外层包裹元素进行计算。如果 `<svg>` 元素为根元素，其百分比根据窗口尺寸计算。3.5 节会展示嵌套 `<svg>` 元素的情况。

3.2 使用默认用户坐标

阅读器设置了一个坐标系统，其中水平坐标（*x* 坐标）向右递增，垂直坐标（*y* 坐标）垂直向下递增。定义视口的左上角 *x* 坐标和 *y* 坐标均为 0。[1] 这个点写作 (0,0)，也被称作原点。这个坐标系统是一个纯粹的几何系统，点没有宽度和高度，其网格线也被认为是无限细的。关于这一主题的更多信息可以参见第 4 章。

示例 3-1 建立了一个 200 像素宽、200 像素高的视口，然后绘制了一个矩形，它的左上角在坐标 (10,10) 位置，宽为 50 像素，高为 30 像素。[2] 图 3-1 展示了其结果，并且使用标尺和网格展示了坐标系统。

注 1：本书中，坐标被指定为括号内的数字对，其中第一个为 *x* 坐标。因此，(10, 30) 表示 *x* 坐标为 10，*y* 坐标为 30。

注 2：为了节省空间，我们省略了 `<?xml ...?>` 和 `<!DOCTYPE ...>` 行。这些内容在每个图形中都是一成不变的，因此忽略它们。

示例 3-1：使用默认的坐标

http://oreillymedia.github.io/svg-essentials-examples/ch03/default_coordinates.html

```
<svg width="200" height="200">
    <rect x="10" y="10" width="50" height="30"
        style="stroke: black; fill: none;"/>
</svg>
```

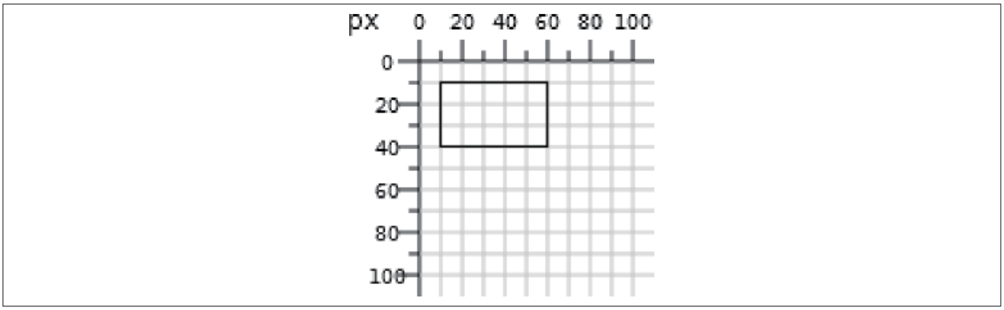


图 3-1：使用默认坐标的矩形

即使视口中没有指定单位，也可以在某些 SVG 形状元素中指定单位，如示例 3-2 所示。图 3-2 展示了其结果，并使用标尺和网格展示了坐标系统。

示例 3-2：明确指定单位

http://oreillymedia.github.io/svg-essentials-examples/ch03/explicit_units.html

```
<svg width="200" height="200">
    <rect x="10mm" y="10mm" width="15mm" height="10mm"
        style="stroke: black; fill: none;"/>
</svg>
```

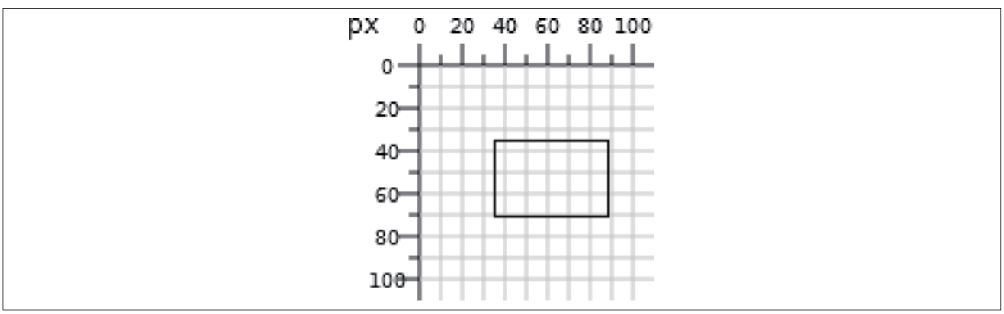


图 3-2：使用明确指定单位的矩形

在 <svg> 元素中指定单位并不会影响其他元素中没有给定单位的坐标。示例 3-3 展示了一个以毫米为单位进行设置的视口，而矩形仍然使用像素（用户）坐标绘制，正如图 3-3 所示。

示例 3-3：<svg> 元素上的单位

http://oreillymedia.github.io/svg-essentials-examples/ch03/units_on_svg.html

```
<svg width="70mm" height="70mm">
    <rect x="10" y="10" width="50" height="30"
        style="fill:none; stroke:black;"/>
</svg>
```

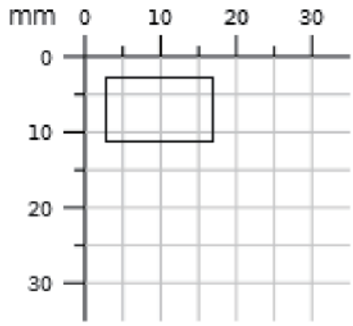


图 3-3：指定单位的视口和没有指定单位的矩形

3.3 为视口指定用户坐标

到目前为止的示例中，没有单位的数值都被视为像素。有时候这并不是我们想要的。比如，你可能想要设置一个坐标系统，其中每个用户坐标表示 1/16 厘米（这并不是一个好的设计，我们这么用只是为了证明确实可行）。在这个系统中，一个边长为 40 单位的方块显示的边长为 2.5 厘米。

为了实现这一效果，我们将在 <svg> 元素上设置 viewBox 属性。这个属性的值由 4 个数值组成，它们分别代表想要叠加在视口上的用户坐标系统的最小 x 坐标、最小 y 坐标、宽度和高度。

因此，要在 4 厘米 ×5 厘米的图纸上设置一个每厘米 16 个单位的坐标系统，要使用这个开始标记：

```
<svg width="4cm" height="5cm" viewBox="0 0 64 80">
```

示例 3-4 给出了一个房子图像的 SVG，并且使用新的坐标系统显示。图 3-4 展示了结果。其中网格和深色的数值展示了新的用户坐标系统，浅色的数值之间间隔 1 厘米。

示例 3-4：使用 viewBox

http://oreillymedia.github.io/svg-essentials-examples/ch03/using_viewbox.html

```
<svg width="4cm" height="5cm" viewBox="0 0 64 80">
```

```
    <rect x="10" y="35" width="40" height="40"
      style="stroke: black; fill: none;"/>
    <!-- 房顶 -->
    <polyline points="10 35, 30 7.68, 50 35"
      style="stroke: black; fill: none;"/>
    <!-- 房门 -->
    <polyline points="30 75, 30 55, 40 55, 40 75"
      style="stroke: black; fill: none;"/>
</svg>
```

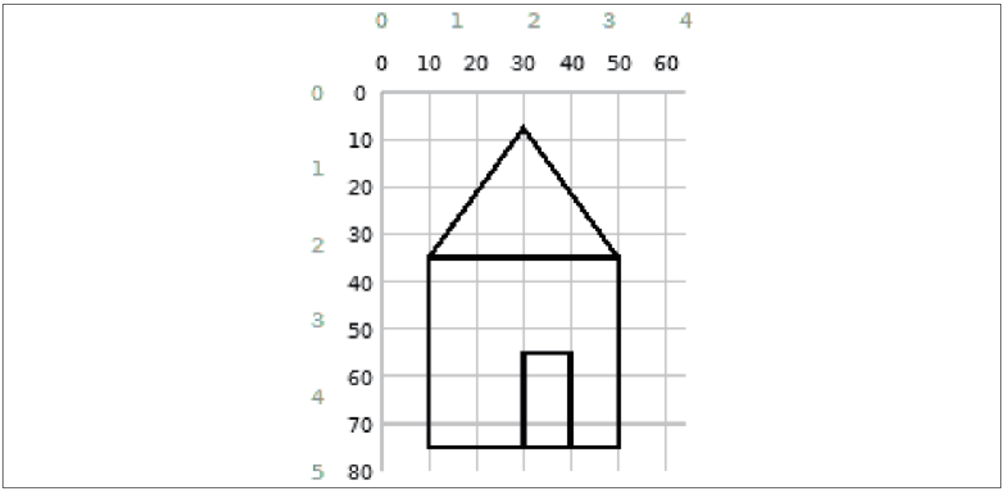


图 3-4：新的用户坐标系统

为 viewBox 属性指定的数值可以使用逗号或空格分隔。如果宽度或者高度为 0，则没有图形显示。宽度和高度要求大于等于 0，为它们指定负值是错误的。

> 仔细阅读示例 3-4 的代码，你会注意到，为了让房顶的位置更精确，我们使用了十进制的小数来表示它。在 SVG 中几乎所有的数字都是十进制浮点数。SVG 阅读器程序需要支持至少 32 位精度的数字，并且对于某些计算鼓励使用更高精度的数字。实际上，我们甚至可以在坐标系统中使用科学计数法来表示非常大或者非常小的数字，因此点 30,7.68 还可以写作 3.0E+1,7.68e0。但为了可读性和简洁性，并不推荐采用这种形式——仅在必要时再采用吧。

3.4 保留宽高比

在前面的例子中，视口和 viewBox 的宽高比是相同的（4/5 = 64/80）。但是，如果视口的宽高比和 viewBox 不一样，会发生什么情况？如同在这个示例中，viewBox 的宽高比为 1 ∶ 1（宽度和高度相同），但是视口的宽高比为 1 ∶ 3（高度是宽度的 3 倍）。

```
<svg width="45px" height="135px" viewBox="0 0 90 90">
```

在这种情况下，SVG 可以做三件事。

- 按较小的尺寸等比例缩放图形，以使图形完全填充视口。在这个例子中，图片将变为原始宽高的一半。在 3.4.2 节会看到相关的例子。
- 按较大的尺寸等比例缩放图形并裁剪掉超出视口的部分。在这个例子中，图片会变成原始宽高的 1.5 倍。在 3.4.3 节会看到相关的例子。
- 拉伸和挤压绘图以使其恰好填充新的视口（也就是说，完全不保留宽高比）。在 3.4.4 节可查看详情。

在第一种情况下，由于图片在某一方向比视口小，所以我们必须指定将图片放置在哪里。在这个例子中，图片的宽高会被统一缩放为 45 像素。缩小后图形的宽度完美适配视口的宽度，但你必须决定图片是显示在 135 像素高的视口的顶部（顶部对齐）、中间还是底部。

在第二种情况下，由于图片在某一方向比视口大，我们必须指定哪个区域被剪切掉。在这个例子中，图片的宽高会被统一缩放为 135 像素。至此，图片的高度非常适合视口，但你必须决定是切掉图片的右侧、左侧还是两侧，以适配 45 像素宽的视口。

3.4.1 为`preserveAspectRatio`指定对齐方式

`preserveAspectRatio` 属性允许我们指定被缩放的图像相对视口的对齐方式，以及是希望它适配边缘还是要裁剪。这一属性的模型为：

```
preserveAspectRatio="alignment [meet | slice]"
```

其中 `alignment` 指定轴和位置，可选值为表 3-1 中的组合值之一。对齐说明符由一个 *x* 对齐方式的值和一个 *y* 对齐方式的值（`min`、`mid` 或者 `max`）组合而成。`preserveAspectRatio` 的默认值为 `xMidYMid meet`。

> *y* 对齐方式由大写字母开始，因为 *x* 对齐方式和 *y* 对齐方式被连接成为一个单词。

表3-1：`preseveAspectRatio`对齐方式的可选值

*y*对齐	*x*对齐		
	`xMin` 按视口左侧边缘，`viewBox`最小*x*值对齐	`xMid` 按视口水平中心，`viewBox`中点*x*值对齐	`xMax` 按视口右侧边缘，`viewBox`最大*x*值对齐
`yMin` 按视口顶部边缘，`viewBox`最小*y*值对齐	`xMinYMin`	`xMidYMin`	`xMaxYMin`

（续）

y对齐	x对齐		
yMid 按视口垂直中心，viewBox中点y值对齐	xMinYMid	xMidYMid	xMaxYMid
yMax 按视口底部边缘，viewBox最大y值对齐	xMinYMax	xMidYMax	xMaxYMax

因此，如果想要 `viewBox="0 0 90 90"` 内的图片完全适配宽为 45 像素、高为 135 像素的视口，并且与视口顶部对齐，要编写如下所示代码：

```
<svg width="45px" height="135px" viewBox="0 0 90 90"
  preserveAspectRatio="xMinYMin meet">
```

在这种情况下，由于宽度正好适配，因此 x 对齐方式并不重要；你也可以使用 `xMidYMin` 或者 `xMaxYMin`。然而，通常在不知道视口宽高比时才会使用 `preserveAspectRatio`。比如，你可能希望缩放图像以适配应用程序窗口，或者可能使用父文档的 CSS 设置高度和宽度。在这些情况下，你需要考虑视口太宽或者太高时如何显示图像。

如果没有指定 `preserveAspectRatio`，其默认值为 `xMidYMid meet`，它会缩小图像以适配可用的空间，并且使它水平和垂直居中。

这些内容都很抽象，下面给出一些具体例子，展示组合对齐方式与 `meet` 和 `slice` 如何交互。

3.4.2 使用meet说明符

示例 3-5 中的 `<svg>` 开始标记都使用了 `meet` 说明符。

示例 3-5：使用 meet 说明符

```
<!-- 高视口 -->
<svg preserveAspectRatio="xMinYMin meet" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMidYMid meet" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMaxYMax meet" viewBox="0 0 90 90"
    width="45" height="135">

<!-- 宽视口 -->
<svg preserveAspectRatio="xMinYMin meet" viewBox="0 0 90 90"
    width="135" height="45">
```

```
<svg preserveAspectRatio="xMidYMid meet" viewBox="0 0 90 90"
    width="135" height="45">

<svg preserveAspectRatio="xMaxYMax meet" viewBox="0 0 90 90"
    width="135" height="45">
```

图 3-5 展示了缩小的图像适配闭合 viewBox 的情况。

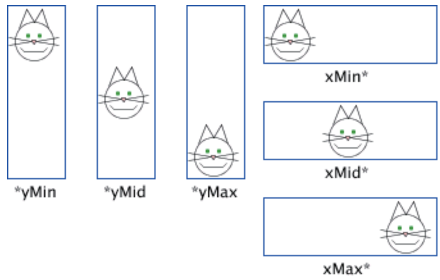


图 3-5：meet—viewBox 适配视口

3.4.3　使用slice说明符

图 3-6 展示了使用 slice 说明符裁剪图像不适合视口的部分。它们都是用示例 3-6 中的 <svg> 标签创建的。

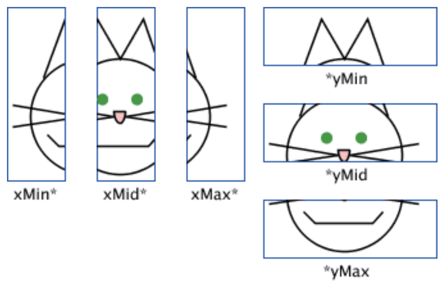


图 3-6：slice——图像填充视口

示例 3-6：使用 slice 说明符

```
<!-- 高视口 -->
<svg preserveAspectRatio="xMinYMin slice" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMidYMid slice" viewBox="0 0 90 90"
    width="45" height="135">

<svg preserveAspectRatio="xMaxYMax slice" viewBox="0 0 90 90"
    width="45" height="135">
```

```
<!-- 宽视口 -->
<svg preserveAspectRatio="xMinYMin slice" viewBox="0 0 90 90"
    width="135" height="45">

<svg preserveAspectRatio="xMidYMid slice" viewBox="0 0 90 90"
    width="135" height="45">

<svg preserveAspectRatio="xMaxYMax slice" viewBox="0 0 90 90"
    width="135" height="45">
```

这一节的在线示例允许你尝试不同的 `preserveAspectRatio` 选项，根据任意大小的 SVG 剪切、缩放和移动猫：

http://oreillymedia.github.io/svg-essentials-examples/ch03/meet_slice_specifier.html

3.4.4　使用none说明符

最后，还有第三个选项用于在 `viewBox` 和视口的宽高比不同时缩放图像。如果指定 `preserveAspectRatio="none"`，那么图像不会被等比例缩放，以使它的用户坐标适合视口。图 3-7 展示了使用示例 3-7 中的 `<svg>` 标签生成的“哈哈镜”效果。

示例 3-7：不保留宽高比

```
<!-- 高视口 -->
<svg preserveAspectRatio="none" viewBox="0 0 90 90"
    width="45" height="135">

<!-- 宽视口 -->
<svg preserveAspectRatio="none" viewBox="0 0 90 90"
    width="135" height="45">
```

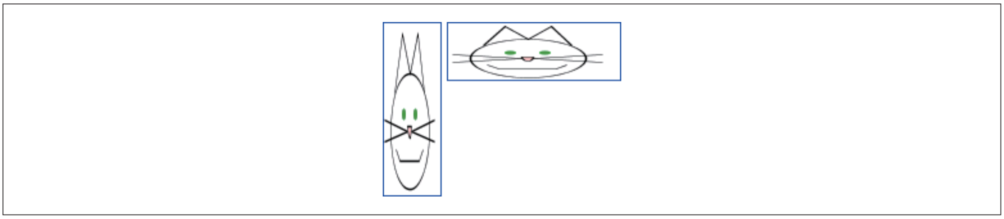


图 3-7：不保留宽高比

3.5　嵌套坐标系统

我们可以在任何时候将另一个 `<svg>` 元素插入到文档中来建立新的视口和坐标系统。其效果是创建一个稍后可以绘制图形的“迷你画布”。图 3-5 就是使用这一技术实现的。我们并没有逐一绘制矩形，然后调整和定位每一个矩形里的猫（暴力的方法），而是采用如下步骤。

- 在主画布上绘制蓝色的矩形。
- 为每个矩形定义一个带有对应 preserveAspectRatio 属性的新 <svg> 元素。
- 在新画布上绘制猫（使用 <use>），并让 SVG 做其他工作。

这里有一个简化的例子，在主画布上展示一个圆，然后在主画布上的新画布（轮廓为一个蓝色矩形）内还展示了一个圆。图 3-8 便是期望的结果。

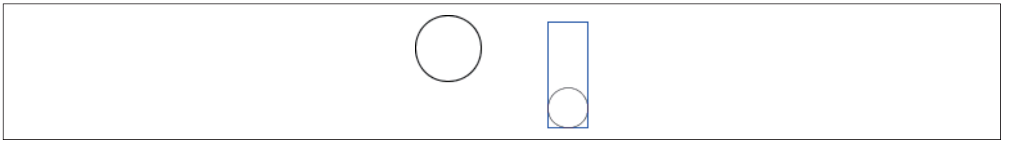


图 3-8：嵌套的视口

首先，为主坐标系统生成 SVG 和圆（注意，其用户坐标与这个文档中的视口正好重合）。

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
    <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
</svg>
```

结果如图 3-9 所示。

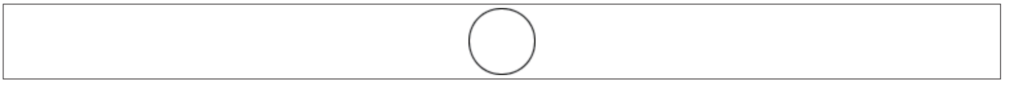


图 3-9：主视口中的圆

现在，绘制盒子的边界来显示新视口的位置：

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
    <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
    <rect x="100" y="5" width="30" height="80"
       style="stroke: blue; fill: none;"/>
</svg>
```

这就产生了如图 3-10 所示的效果。

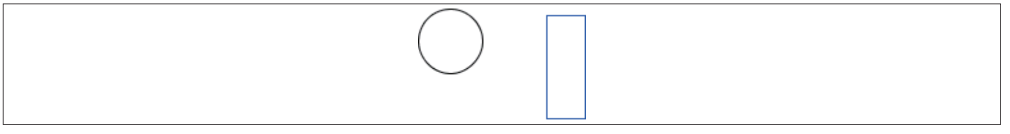


图 3-10：主视口中的圆和边界盒子

现在，我们来为新视口添加另一个 <svg> 元素。除了指定 viewBox、width、height 和 preserveAspectRatio 规格，还可以在闭合的 <svg> 元素上指定 x 和 y 属性，建立新视口（如果没有给 x 和 y 指定值，则假定为 0）。

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
    <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
    <rect x="100" y="5" width="30" height="80"
        style="stroke: blue; fill: none;"/>

    <svg x="100px" y="5px" width="30px" height="80px"
      viewBox="0 0 50 50" preserveAspectRatio="xMaxYMax meet">
    </svg>
</svg>
```

使用嵌套的 <svg> 元素设置新的坐标系统并不会改变视觉显示，但它允许我们在该新系统中添加圆，生成的结果如图 3-8 所示。

```
<svg width="200px" height="200px" viewBox="0 0 200 200">
  <circle cx="25" cy="25" r="25" style="stroke: black; fill: none;"/>
  <rect x="100" y="5" width="30" height="80" style="stroke: blue;
    fill: none;"/>

  <svg x="100px" y="5px" width="30px" height="80px" viewBox="0 0 50 50"
    preserveAspectRatio="xMaxYMax meet">
    <circle cx="25" cy="25" r="25" style="stroke: black;
      fill: none;"/>
  </svg>
</svg>
```

如果你尝试为一个 <svg> 的 preserveAspectRatio 属性使用 meet 或者 slice 值，而这个 <svg> 嵌套在另一个具有 preserveAspectRatio="none" 属性的 <svg> 元素内，其结果可能会让你大吃一惊。嵌套元素的视口的宽高比，将按照父 SVG 被压缩或者拉伸的坐标来求值以适配视口，这可能导致图像被挤压和裁剪或者缩放。

第 4 章

基本形状

一旦在 <svg> 标签中建立起坐标系统，就可以开始画图了。本章将介绍用来创建大部分绘图中主要元素的基本形状：线段、矩形、多边形、圆和椭圆。

4.1 线段

SVG 可以使用 <line> 元素画出一条直线段。使用时只需要指定线段起止点的 *x* 和 *y* 坐标即可。指定坐标时可以不带单位，此时会使用用户坐标，也可以带上单位，如 em、in 等（3.1 节介绍过）。

```
<line x1="start-x" y1="start-y"
  x2="end-x" y2="end-y" />
```

示例 4-1 中的 SVG 画了好几条线段，图 4-1 中的坐标格子不是 SVG 图形的组成部分。

示例 4-1：线段

http://oreillymedia.github.io/svg-essentials-examples/ch04/basic-lines.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- 水平线段 -->
  <line x1="40" y1="20" x2="80" y2="20" style="stroke: black;" />
  <!-- 垂直线段 -->
  <line x1="0.7cm" y1="1cm" x2="0.7cm" y2="2.0cm"
    style="stroke: black;" />
   <!-- 对角线段 -->
   <line x1="30" y1="30" x2="85" y2="85" style="stroke: black;" />
</svg>
```

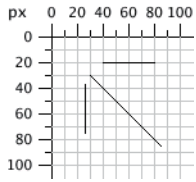


图 4-1：线段

4.2 笔画特性

线段可以看作在画布上画出来的笔画。笔画的尺寸、颜色和风格都会影响线段的表现。这些特性都可以在 style 属性中指定。

4.2.1 stroke-width

第 3 章提到过，画布中的坐标网格线是无穷细的，那么，线段或笔画的位置与网格线的位置是什么关系呢？答案是网格线位于笔画的正中间。示例 4-2 中画了几条线段，为了看得更清晰，将笔画宽度设为 10（用户坐标）。图 4-2 中画出了坐标格子的线，以便你看得更清楚。

示例 4-2：演示 stroke-width

http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-width.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- 水平线段 -->
  <line x1="30" y1="10" x2="80" y2="10"
    style="stroke-width: 10; stroke: black;" />
  <!-- 垂直线段 -->
  <line x1="10" y1="30" x2="20" y2="80"
    style="stroke-width: 10; stroke: black;" />
  <!-- 对角线段 -->
  <line x1="25" y1="25" x2="75" y2="75"
    style="stroke-width: 10; stroke: black;" />
</svg>
```

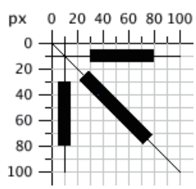


图 4-2：演示 stroke-width

SVG 的坐标网格线可能是无穷细的，但是你的电脑屏幕却是由固定大小的像素组成的。对角线的边缘看起来会很毛躁，这是因为电脑屏幕在显示的时候通过计算将它放到了最邻近的像素块中，这就是所谓的锯齿（aliasing）。电脑也可以使用反锯齿（anti-aliasing）技术来使边缘看起来更柔和，具体做法是对那些斜线只经过了一部分的像素点进行模糊处理。

大部分 SVG 阅读器都会默认开启反锯齿功能，这会使得 1 像素的黑线有时候看起来看 2 像素的灰线，因为它位于屏幕上两个像素的正中间。你可以通过指定 CSS 属性 `shape-rendering` 的值来控制反锯齿特性。取值 `crispEdges`（在元素上，或者在整个 SVG 上）会关闭反锯齿特性，得到清晰的图像（有时候看起来很毛躁）。取值 `geometricPrecision` 则会使边缘圆滑（有时候看起来很模糊）。

4.2.2　笔画颜色

你可以通过以下几种方式指定笔画颜色。

- 基本的颜色关键字：`aqua`、`black`、`blue`、`fuchsia`、`gray`、`green`、`lime`、`maroon`、`navy`、`olive`、`purple`、`red`、`silver`、`teal`、`white` 和 `yellow`。也可以使用 SVG 规范第 4.2 节中规定的颜色关键字（http://www.w3.org/TR/SVG/types.html#ColorKeywords）。
- 由 6 位十六进制数字指定的颜色，形式为 `#rrggbb`，其中 `rr` 表示红色，`gg` 表示绿色，`bb` 表示蓝色，它们的范围都是 `00-ff`。
- 由 3 位十六进制数字指定的颜色，形式为 `#rgb`，其中 `r` 表示红色，`g` 表示绿色，`b` 表示蓝色，它们的范围都是 `0-f`。这是前一种方式的简写，与之等效的 6 位数字写法是将每位重复一次，也就是 `#d6e` 与 `#dd66ee` 是一样的。
- 通过 `rgb(red-value, green-value, blue-value)` 形式指定的 `rgb` 颜色值，每个值的取值范围是整数 `0-255` 或者百分比 `0%-100%`。
- `currentColor` 关键字，表示当前元素应用的 CSS 属性 `color` 的值。`color` 是用来给 HTML 的文本设置颜色的，会被子元素继承，但对 SVG 没有直接效果。如果使用内联 SVG 图标的话（见 2.3.2 节），使用 `currentColor` 可以让图标使用它周围文本的颜色。

示例 4-3 使用了上面介绍的各种方法（除了 `currentColor`）指定颜色，结果见图 4-3。

示例 4-3：笔画颜色

http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-color.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- 红色 -->
  <line x1="10" y1="10" x2="50" y2="10"
    style="stroke: red; stroke-width: 5;"/>
```

```
  <!-- 淡绿色 -->
  <line x1="10" y1="20" x2="50" y2="20"
    style="stroke: #9f9; stroke-width: 5;"/>

  <!-- 淡蓝色 -->
  <line x1="10" y1="30" x2="50" y2="30"
    style="stroke: #9999ff; stroke-width: 5;"/>

  <!-- 橘色 -->
  <line x1="10" y1="40" x2="50" y2="40"
    style="stroke: rgb(255, 128, 64); stroke-width: 5;"/>

  <!-- 深紫色 -->
  <line x1="10" y1="50" x2="50" y2="50"
    style="stroke: rgb(60%, 20%, 60%); stroke-width: 5;"/>
</svg>
```



图 4-3：笔画颜色

还有更多指定颜色的方法，它们来自 CSS3 颜色规范（http://www.w3.org/TR/css3-color/）。尽管它们在浏览器中被广泛支持，但并不属于 SVG1.1 规范，可能不被其他的 SVG 实现所支持。比如，在写作本书时，Apache Batik 和 Inkscape 都不支持。这些方法共有三个新颜色函数和一个新关键词。

- `rgba()`，形式为 `rgba(red-value, green-value, blue-value, alpha-value)`，其中颜色值的取值和 `rgb()` 函数一样，透明度（alpha）的取值范围是 0~1。
- `hsl()`，形式为 `hsl(hue, saturation, lightness)`，其中色相值（hue）是整数角度，取值是 0~360，饱和度（saturation）和明度（lightness）的取值范围为整数 0~255 或者百分比 0%~100%。
- `hsla()`，色相、饱和度、明度取值与 `hsl()` 一样，透明度取值与 `rgba()` 一样。
- `transparent`，完全透明，等价于 `rgba(0,0,0,0)`。

如果不指定笔画颜色的话，将看不到任何线，因为 `stroke` 属性的默认值是 `none`。

4.2.3 stroke-opacity

到目前为止，示例中所有的线都是实线，会遮住任何在其下面的东西。我们可以通过给 `stroke-opacity` 属性赋值来控制线条的不透明度（opacity，与透明度相反），取值范围为

0.0~1.0，其中 0 代表完全透明，1 代表完全不透明。小于 0 的值会被更改为 0，大于 1 的值会被更改为 1。示例 4-4 演示了以 0.2 为差值，不透明度从 0.2 到 1 的变化，结果如图 4-4。图中加了红色线，是为了让你更清晰地看到透明度的不同。

示例 4-4：演示 stroke-opacity

http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-opacity.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <line x1="30" y1="0" x2="30" y2="60"
        style="stroke: red; stroke-width: 5;"/>
  <line x1="10" y1="10" x2="50" y2="10"
    style="stroke-opacity: 0.2; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="20" x2="50" y2="20"
    style="stroke-opacity: 0.4; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="30" x2="50" y2="30"
    style="stroke-opacity: 0.6; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="40" x2="50" y2="40"
    style="stroke-opacity: 0.8; stroke: black; stroke-width: 5;"/>
  <line x1="10" y1="50" x2="50" y2="50"
    style="stroke-opacity: 1.0; stroke: black; stroke-width: 5;"/>
</svg>
```



图 4-4：演示 stroke-opacity

4.2.4 stroke-dasharray属性

如果你需要点线或虚线，则需要使用 stroke-dasharray 属性，它的值由一列数字构成，代表线的长度和空隙的长度，数字之间用逗号或空格分隔。数字的个数应该为偶数，但如果你指定的数字个数为奇数，则 SVG 会重复一次，使得总个数为偶数。（见示例 4-5 的最后一个实例。）

示例 4-5：演示 stroke-dasharray 属性

http://oreillymedia.github.io/svg-essentials-examples/ch04/stroke-dasharray.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
   <!-- 9个像素的虚线,5个像素的空隙 -->
   <line x1="10" y1="10" x2="100" y2="10"
       style="stroke-dasharray: 9, 5;
       stroke: black; stroke-width: 2;"/>

   <!-- 5个像素的虚线,3个像素的空隙,9个像素的虚线,2个像素的空隙 -->
   <line x1="10" y1="20" x2="100" y2="20"
```

```
        style="stroke-dasharray: 5, 3, 9, 2;
        stroke: black; stroke-width: 2;"/>

    <!-- 复制奇数个数,这等价于:9个像素的虚线,3个像素的空隙,
         5个像素的虚线,9个像素的空隙,3个像素的虚线,5个像素的空隙 -->
    <line x1="10" y1="30" x2="100" y2="30"
        style="stroke-dasharray: 9,3,5;
        stroke: black; stroke-width: 2;"/>
</svg>
```

结果见图 4-5，为了看得更清晰，此处进行了放大。

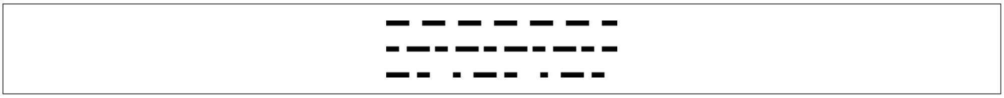


图 4-5：演示 stroke-dasharray

4.3　矩形

矩形是最简单的基本形状，只需要指定其左上角的 x 和 y 坐标[1] 以及它的宽度（width）和高度（height）即可。矩形内部会使用 fill 属性代表的颜色进行填充，如果没有指定 fill 颜色，则会使用黑色填充。fill 属性的值可以用 4.2.2 节所描述的任何一种方式指定，也可以指定为 none，即不填充矩形内部，保持透明。也可以通过与 stroke-opacity 一样的方式（参见 4.2.3 节）来指定 fill-opacity。fill 和 fill-opacity 都是表现属性，属于 style 属性的值。

填充完矩形内部（如果需要的话）之后，接下来会绘制矩形的边框。绘制过程中可以指定的特性，与上文中画线时指定的 stroke 一样。如果不指定 stroke，则它的值为 none，将不会绘制矩形边框。示例 4-6 中绘制了好几个不同的 <rect> 元素。图 4-6 展示了结果，为了看起来更清晰，带上了坐标网格。

示例 4-6：矩形

http://oreillymedia.github.io/svg-essentials-examples/ch04/rectangle.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- 内部填充为黑色,不绘制边框 -->
  <rect x="10" y="10" width="30" height="50"/>

  <!-- 内部不填充颜色,绘制黑色边框 -->
  <rect x="50" y="10" width="20" height="40"
```

注 1：从技术上讲，x 的值是指矩形的“左右”角在当前用户坐标中的较小的 x 值，y 的值是指矩形的“上下”角在当前用户坐标中的较小的 y 值。因为你还没有用到变换（第 6 章介绍），所以此处使用更易理解的“左上角”。

```
    style="fill: none; stroke: black;"/>

  <!-- 内部填充为蓝色,绘制较粗的、半透明红色边框 -->
  <rect x="10" y="70" width="25" height="30"
    style="fill: #0000ff;
      stroke: red; stroke-width: 7; stroke-opacity: 0.5;"/>

  <!-- 内部填充为半透明的黄色,用虚线绘制绿色边框 -->
  <rect x="50" y="70" width="35" height="20"
    style="fill: yellow; fill-opacity: 0.5;
      stroke: green; stroke-width: 2; stroke-dasharray: 5 2"/>
</svg>
```

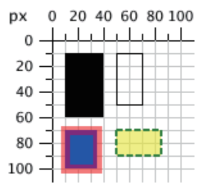


图 4-6：矩形

构成边框的笔画是“骑”在坐标网格线上的，所以一半在矩形内，一半在矩形外。通过图 4-7 中的特写能更清晰地看到示例 4-6 中的红色半透明的边框。

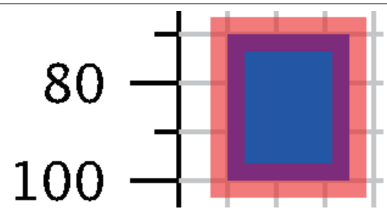


图 4-7：半透明边框的特写

矩形坐标的 x 和 y 值默认都为 0。如果指定宽或高（width/height）为 0，则矩形不显示。如果指定宽或高为负值会报错。

圆角矩形

如果你希望得到一个圆角矩形，可以分别指定 *x* 方向和 *y* 方向的圆角半径。rx（x-radius，*x* 方向的圆角半径）的最大值是矩形宽度的一半，同理，ry（y-radius，y 方向的圆角半径）的最大值是矩形高度的一半。如果只指定了 rx 和 ry 中的一个值，则认为它们相等。示例 4-7 展示了 rx 和 ry 的用法。

示例 4-7：圆角矩形

http://oreillymedia.github.io/svg-essentials-examples/ch04/rounded-rectangles.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- rx和ry相等,逐渐增大 -->
  <rect x="10" y="10" width="20" height="40" rx="2" ry="2"
    style="stroke: black; fill: none;"/>

  <rect x="40" y="10" width="20" height="40" rx="5"
    style="stroke: black; fill: none;"/>

  <rect x="70" y="10" width="20" height="40" ry="10"
    style="stroke: black; fill: none;"/>

  <!-- rx和ry不相等 -->
  <rect x="10" y="60" width="20" height="40" rx="10" ry="5"
    style="stroke: black; fill: none;"/>

  <rect x="40" y="60" width="20" height="40" rx="5" ry="10"
    style="stroke: black; fill: none;"/>
</svg>
```

图 4-8 显示了结果，为了看起来更清晰，带上了坐标网格。

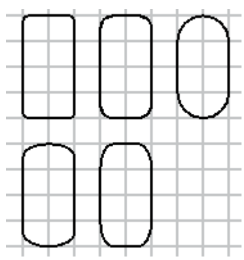


图 4-8：圆角矩形



如果你熟悉 CSS 的 border-radius 属性，可能知道通过设置圆角半径为宽高的 50%，可使矩形变成一个圆或椭圆。在 SVG 中你也可以通过百分比来指定圆角半径的值，但会被解析为相对视口的宽高的百分比（和使用百分比来设置矩形的宽或高一样），而不是相对矩形本身的宽高的百分比。不过好在要创建圆和椭圆的话，SVG 有更简单的方法。

4.4　圆和椭圆

要画一个圆，需要使用 <circle> 元素，并指定圆心的 *x* 和 *y* 坐标（cx/cy）以及半径（r）。和矩形一样，不指定 fill 和 stroke 的情况下，圆会使用黑色填充并且没有轮廓线。

椭圆除了需要指定圆心的坐标外，还需要同时指定 *x* 方向的半径和 *y* 方向的半径，属性分别是 rx 和 ry。

对圆和椭圆来说，如果省略 cx 或者 cy，则默认为 0。如果半径为 0，则不会显示图形；如果半径为负数，则会报错。示例 4-8 画了一些圆和椭圆，结果如图 4-9。

示例 4-8：圆和椭圆

http://oreillymedia.github.io/svg-essentials-examples/ch04/circles-ellipses.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <circle cx="30" cy="30" r="20" style="stroke: black; fill: none;"/>
  <circle cx="80" cy="30" r="20"
    style="stroke-width: 5; stroke: black; fill: none;"/>

  <ellipse cx="30" cy="80" rx="10" ry="20"
    style="stroke: black; fill: none;"/>
  <ellipse cx="80" cy="80" rx="20" ry="10"
    style="stroke: black; fill: none;"/>
</svg>
```

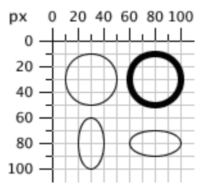


图 4-9：圆和椭圆

4.5 多边形

除了矩形、圆和椭圆，你可能还想画六边形、八边形、五角星或者其他封闭图形。<polygon> 元素可以用来画任意封闭图形，这个图形也可以像前面说的那样填充和绘制轮廓。使用时需要为 points 属性指定一系列的 *x*/*y* 坐标对，并用逗号或者空格分隔。表示坐标的数字个数必须是偶数。指定坐标时不需要在最后指定返回起始坐标，因为图形是封闭的，会自动回到起始坐标。示例 4-9 演示了使用 <polygon> 绘制一个平行四边形、一个五角星和一个不规则图形。

示例 4-9：多边形

http://oreillymedia.github.io/svg-essentials-examples/ch04/polygon.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- 平行四边行 -->
  <polygon points="15,10  55, 10  45, 20  5, 20"
```

```
    style="fill: red; stroke: black;"/>

  <!-- 五角星 -->
  <polygon
    points="35,37.5  37.9,46.1 46.9,46.1  39.7,51.5
      42.3,60.1  35,55  27.7,60.1  30.3,51.5
      23.1,46.1  32.1,46.1"
      style="fill: #ccffcc; stroke: green;"/>

  <!-- 不规则图形 -->
  <polygon
    points="60 60,  65 72,  80 60,  90 90, 72 80, 72 85, 50 95"
    style="fill: yellow; fill-opacity: 0.5; stroke: black;
      stroke-width: 2;"/>
</svg>
```

结果如图 4-10，为了看起来更清晰，带上了坐标网格。

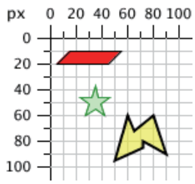


图 4-10：多边形

填充边线交叉的多边形

到目前为止，我们看到的多边形都很容易填充，因为多边形的各边都没有交叉，很容易区分出多边形的内部区域和外部区域。但是，当多边形的边彼此交叉的时候，要区分哪些区域是图形内部并不容易。示例 4-10 使用 SVG 画了一个这样的图形，在图 4-11 中，中间的那部分到底要算图形内部还是图形外部呢？

示例 4-10：未填充的边线交叉的多边形

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">

<polygon  points="48,16 16,96 96,48 0,48 80,96"
  style="stroke: black; fill: none;"/>

</svg>
```



图 4-11：未填充的边线交叉的多边形

SVG 有两种判断某个点是否在多边形中的规则。分别对应 `fill-rule`（表现的一部分）属性的 `nonzero`（默认值）和 `evenodd`。选择不同的规则会有不同的效果。示例 4-11 使用了这两种规则来填充五角星，结果如图 4-12。

示例 4-11：不同填充规则的效果

http://oreillymedia.github.io/svg-essentials-examples/ch04/polygon-fill.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">

<polygon style="fill-rule: nonzero; fill: yellow; stroke: black;"
  points="48,16 16,96 96,48 0,48 80,96"/>

<polygon style="fill-rule: evenodd; fill: #00ff00; stroke: black;"
  points="148,16 116,96 196,48 100,48 180,96"/>

</svg>
```

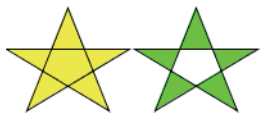


图 4-12：不同填充规则的效果

> **填充规则的解释**
>
> 为了保持完整性，还是解释一下 `fill-rules` 的原理，但是你在使用时并不一定要知道这些细节。`nonzero` 规则在判断某个点是否在图形内部时，从这个点画一条线到无穷远，然后数这条线与图形边线有多少次交叉。如果交叉的边线是从右往左画，则总数加 1；如果交叉的边线是从左往右画，则总数减 1。如果最后总数为 0，则认为该点在图形外部，否则认为在图形内部。
>
> `evenodd` 规则也画了同样一条线，但它只算与边线相交的次数。如果总数是奇数，则认为点在图形内部，否则认为点在图形外部。

4.6 折线

最后，为了使我们关于基本形状的讨论更完整，我们回归到直线。有时候你希望有一系列的直线段，但并不闭合为某个形状。你可以使用多个 `<line>` 元素，但如果有非常多线的话，可能使用 `<polyline>` 元素会更容易。它与 `<polygon>` 有相同的 `points` 属性，不同之处在于图形并不封闭。示例 4-12 画了一个电阻的符号，结果见图 4-13。

示例 4-12：折线

http://oreillymedia.github.io/svg-essentials-examples/ch04/polyline.html

```
<svg width="100px" height="50px" viewBox="0 0 100 50"
  xmlns="http://www.w3.org/2000/svg">

<polyline
  points="5 20, 20 20, 25 10, 35 30, 45 10,
    55 30, 65 10, 75 30, 80 20, 95 20"
  style="stroke: black; stroke-width: 3; fill: none;"/>
</svg>
```

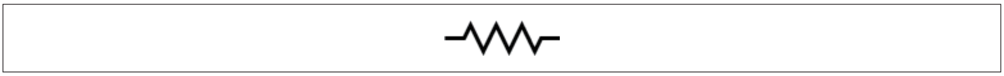


图 4-13：折线

在使用 `<polyline>` 时最好将 `fill` 属性设为 `none`，否则 SVG 阅读器会尝试去填充形状，有时候会意外地看到如图 4-14 所示的情况。

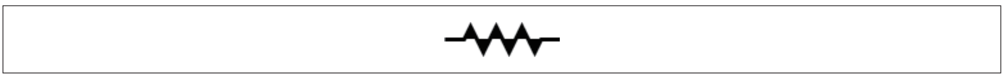


图 4-14：填充的折线

4.7 线帽和线连接

当使用 `<line>` 或者 `<polyline>` 画线时，可以为 `stroke-linecap` 指定不同的值来确定线的头尾形状，可能的取值为 `butt` 、`round`、`square`。示例 4-13 中使用了这三个值，并加上了灰色线表示头尾的实际位置。从图 4-15 中可以看到，`round` 和 `square` 在起止位置都超过了真实位置，默认值 `butt` 则精确地与起止位置对齐。

示例 4-13：`stroke-linecap` 属性的不同值

http://oreillymedia.github.io/svg-essentials-examples/ch04/linecap.html

```
<line x1="10" y1="15" x2="50" y2="15"
  style="stroke: black; stroke-linecap: butt; stroke-width: 15;"/>

<line x1="10" y1="45" x2="50" y2="45"
  style="stroke: black; stroke-linecap: round; stroke-width: 15;"/>

<line x1="10" y1="75" x2="50" y2="75"
  style="stroke: black; stroke-linecap: square; stroke-width: 15;"/>
```

```
<!-- 灰色线 -->
<line x1="10" y1="0" x2="10" y2="100" style="stroke: #999;" />
<line x1="50" y1="0" x2="50" y2="100" style="stroke: #999;" />
```

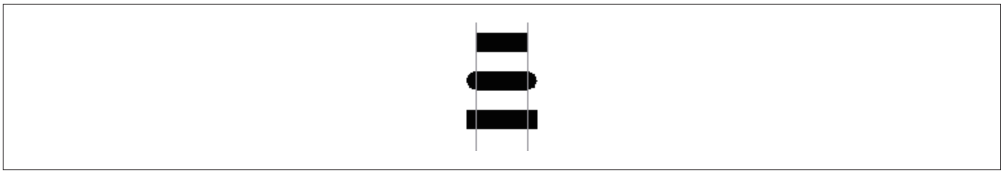


图 4-15：stroke-linecap 属性的不同值

你也可以通过 stroke-linejoin 属性来指定线段在图形棱角处交叉时的效果，可能的取值为 miter（尖的）、round（圆的）、bevel（平的）。示例 4-14 演示了这些不同的值，结果见图 4-16。

示例 4-14：stroke-linejoin 属性的不同值

http://oreillymedia.github.io/svg-essentials-examples/ch04/linejoin.html

```
<polyline
  style="stroke-linejoin: miter; stroke: black; stroke-width: 12;
  fill: none;"
  points="30 30, 45 15, 60 30"/>

<polyline
  style="stroke-linejoin: round; stroke: black; stroke-width: 12;
  fill: none;"
  points="90 30, 105 15, 120 30"/>

<polyline
  style="stroke-linejoin: bevel; stroke-width: 12; stroke: black;
  fill: none;"
  points="150 30, 165 15, 180 30"/>
```

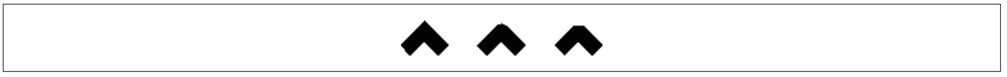


图 4-16：stroke-linejoin 属性的不同值

如果两条线相交时是锐角，且 stroke-linejoin 的值为 miter，则相交处有可能比线本身要宽。你可以指定 stroke-miterlimit 的值来设置相交处的显示宽度与线宽的比率，它的默认值为 4。

4.8 基本形状总结

下面的表格总结了 SVG 中的基本形状以及它们的表现风格。

4.8.1 形状元素

表 4-1 总结了 SVG 中的基本形状。

表4-1：SVG中的基本形状

形　状	描　述
`<line x1="start-x" y1="start-y" x2="end-x" y2="end-y" />`	从起始点 (start-x, start-y) 画一条线到 (end-x, end-y)
`<rect x="left-x" y="top-y" width="width" height="height" />`	画一个矩形，左上角位于 (left-x, top-y)，宽高分别为 width 和 height
`<circle cx="center-x" cy="center-y" r="radius" />`	以指定半径 radius 画一个圆，圆心位于 (center-x, center-y)
`<ellipse cx="center-x" cy="center-y" rx="x-radius" ry="y-radius" />`	画一个椭圆，*x* 方向半径为 x-radius，*y* 方向半径为 y-radius，圆心位于 (center-x, center-y)
`<polygon points="points-list" />`	画一个封闭图形，轮廓由 points-list 指定，它由一系列 *x*/*y* 坐标对组成。这些数值只能使用用户坐标，不可以添加长度单位
`<polyline points="points-list" />`	画一系列相连的折线段，折线点由 points-list 指定，它由一系列 *x*/*y* 坐标对组成。这些数值只能使用用户坐标，不可以添加长度单位

当你为属性指定数值时，默认会以用户坐标为准。表 4-1 中除了最后两个元素以外，其他元素的属性都可以在数字后加上单位，比如 mm、pt 等。如：

```
<line x1="1cm" y1="30" x2="50" y2="10pt" />
```

4.8.2 指定颜色

可以通过以下几种方式指定填充或者轮廓的颜色。

- none，表示不绘制轮廓，或者不为形状填充颜色。
- 基本颜色名称，aqua、black、blue、fuchsia、gray、green、lime、maroon、navy、olive、purple、red、silver、teal、white 和 yellow。
- SVG 规范中规定的扩展颜色名称（http://www.w3.org/TR/SVG/types.html#ColorKeywords）。
- 6 位十六进制数字 #rrggbb，从前往后，每两位分别表示红、绿、蓝色值。
- 3 位十六进制数字 #rgb，3 位数字分别表示红、绿、蓝色值。这是前一种方法的缩写，每位数字重复一次的话，#rgb 和 #rrggbb 等价。
- rgb(r, g, b)，每个值的范围为 0~255 或者 0%~100%。
- currentColor，来自元素的（当前应用的）color 属性，一般是继承的。
- CSS3 颜色模块（http://www.w3.org/TR/css3-color/）规范规定的其他方法（可能不能被所有的 SVG 实现支持）。

4.8.3 笔画和填充特性

为了使线或者轮廓显示出来，必须使用以下属性指定笔画的特性。图形的轮廓会在内部填充完之后进行绘制。表 4-2 中总结的所有特性都是表现特性，都属于 `style` 属性。

表4-2：笔画特性

属　　性	值
`stroke`	笔画颜色，使用 4.8.2 节中的方法指定，默认值为 `none`
`stroke-width`	笔画宽度，可用用户坐标或者指定单位的方式指定。笔画的宽度会相对坐标网格线居中。默认值为 1
`stroke-opacity`	数字，从 0.0 到 1.0。0.0 是完全透明，1.0 是完全不透明（默认值）
`stroke-dasharray`	用一系列的数字来指定虚线和间隙的长度。这些数字只能使用用户坐标，默认值为 `none`
`stroke-linecap`	线头尾的形状，值为 `butt`（默认值）、`round` 或 `square`
`stroke-linejoin`	图形的棱角或者一系列连线的形状，取值为 `miter`（尖的，默认值）、`round` 或者 `bevel`（平的）
`stroke-miterlimit`	相交处显示宽度与线宽的最大比例，默认值为 4

你可以使用表 4-3 中的属性来控制图形内部填充的方式。图形会首先填充内部再绘制轮廓。

表4-3：填充特性

属　　性	值
`fill`	按 4.8.2 节中描述的方式指定填充颜色，默认值为 `black`
`fill-opacity`	从 0.0 到 1.0 的数字，0.0 表示完全透明，1.0（默认值）表示完全不透明
`fill-rule`	属性值为 `nonzero`（默认值）或 `evenodd`。该属性决定判断某个点是否在图形内部的方法。只有当边线交叉时或者内部有“洞”时才有效。详细描述见 4.5 节

这些只是 SVG 元素上可以应用的一部分样式属性，附录 B 中的表 B-1 是完整列表。

第 5 章

文档结构

前面曾提到过，SVG 允许我们将文档表现与文档结构分离。本章，我们将对比文档结构和文档表现，详细讨论文档的表现，然后展示一些可以使文档结构更清晰、更可读以及更容易维护的 SVG 元素。

5.1 结构和表现

正如 1.4.2 节提到的，XML 的目标之一便是提供一种能将结构从视觉表示中独立出来的方法。考虑一下第 1 章中绘制的猫，我们根据它的结构——各种几何图形的位置和尺寸——认出它是一只猫。如果我们改变结构，比如缩短胡须、把鼻子变圆、将耳朵变圆变长，那么不管它看起来是什么样，绘图都会变成一只兔子。也就是说，结构会告诉你图形是什么。

这并不是说视觉样式信息不重要。如果我们绘制紫色线条和灰色填充的猫，你也能认出它是一只猫，只是长得不那么好看。其区别如图 5-1 所示。XML 鼓励我们分离结构和表现，但不幸的是，关于 XML 的很多讨论都强调结构而非表现。我们将通过详细讨论如何在 SVG 中指定表现来纠正这一错误。

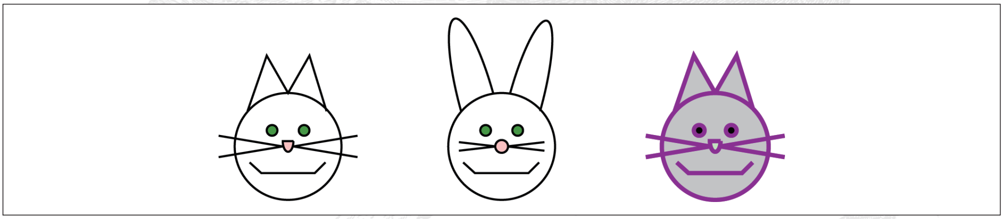


图 5-1：结构与表现

5.2 在SVG中使用样式

SVG 允许我们使用四种方式指定图形表现方面的信息：内联样式、内部样式表、外部样式表以及表现属性。让我们来依次看看每一种方式。

5.2.1 内联样式

示例 5-1 使用了内联样式。到目前为止我们都是使用这种方式指定表现信息，我们设置 `style` 属性的值为一系列视觉属性，它们的值如附录 B 所述。

示例 5-1：使用内联样式

```
<circle cx="20" cy="20" r="10"
    style="stroke: black; stroke-width: 1.5; fill: blue;
           fill-opacity: 0.6"/>
```

5.2.2 内部样式表

可以通过一个内部样式表来罗列常用的样式，而无需在每个 SVG 元素内植入样式。这样可以为所有某一类元素应用样式，也可以使用命名类为特定元素应用样式。示例 5-2 建立了一个内部样式表，这将会为所有的圆绘制一条蓝色双倍粗的虚线并使用浅黄色填充内部。内部样式表被定义在 `<defs>` 元素内，稍后将在本章详细讨论。

这个例子中还绘制了好几个圆。图 5-2 第二行中的圆使用内联样式覆盖了内部样式表中的声明。

示例 5-2：使用内部样式表

http://oreillymedia.github.io/svg-essentials-examples/ch05/internal-stylesheets.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
<defs>
<style type="text/css"><![CDATA[
    circle {
        fill: #ffc;
        stroke: blue;
        stroke-width: 2;
        stroke-dasharray: 5 3;
    }
]]></style>
</defs>

<circle cx="20" cy="20" r="10"/>
<circle cx="60" cy="20" r="15"/>
<circle cx="20" cy="60" r="10" style="fill: #cfc"/>
<circle cx="60" cy="60" r="15"
  style="stroke-width: 1; stroke-dasharray: none;"/>
</svg>
```

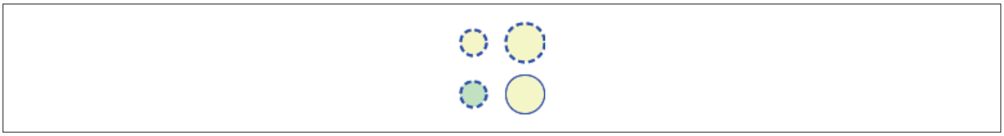


图 5-2：SVG 中的内部样式表

5.2.3　外部样式表

如果想要为多个 SVG 文档应用一组样式，可以通过为每个 SVG 元素复制和粘贴内部样式表的方式来实现。但是如果你希望能统一修改这些文档的样式，则会发现这种方法是不切实际的。我们应该把开始和结束 <style> 标签（不包括 <![CDATA[ ]]>）之间的所有样式信息保存在一个外部文件中，然后把它变成一个外部样式表。示例 5-3 展示了保存在命名为 ext_style.css 的文件中的外部样式表。这个样式表使用了多种选择器，包括 *，它为 SVG 元素内所有没有任何其他样式的元素设置了一个默认样式。生成的结果如图 5-2 所示。

示例 5-3：外部样式表

```
* {fill: none; stroke: black; } /* 所有元素默认样式 */

rect { stroke-dasharray: 7 3; }

circle.yellow { fill: yellow; }

.thick { stroke-width: 5; }

.semiblue { fill: blue; fill-opacity: 0.5; }
```

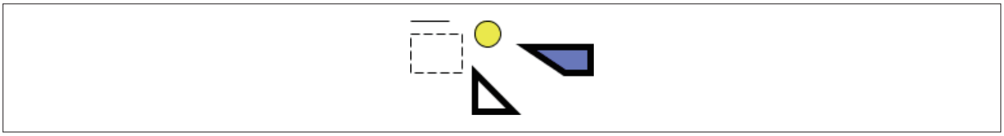


图 5-3：SVG 中的外部样式表

示例 5-4 展示了引用外部样式表的完整 SVG 文档（包含 <?xml ...?>、<?xml-stylesheet ...?> 和 <!DOCTYPE>）。

示例 5-4：引用外部样式表的 SVG 文件

```
<?xml version="1.0"?>
<?xml-stylesheet href="ext_style.css" type="text/css"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
  width="200px" height="200px" viewBox="0 0 200 200">

<line x1="10" y1="10" x2="40" y2="10"/>
<rect x="10" y="20" width="40" height="30"/>
```

```
<circle class="yellow" cx="70" cy="20" r="10"/>
<polygon class="thick" points="60 50, 60 80, 90 80"/>
<polygon class="thick semiblue"
  points="100 30, 150 30, 150 50, 130 50"/>
</svg>
```

内联样式几乎总是比内部或者外部样式表渲染得更快，这是因为样式表和类增加了渲染时的查询和解析时间。然而样式表更容易管理，更小的文件体积和可缓存的特性可以加快文档加载速度。

5.2.4　表现属性

虽然绝大多数 SVG 文档都使用样式来表达表现信息，但 SVG 的确允许我们使用表现属性指定这些信息。我并不是指像这样指定：

```
<circle cx="10" cy="10" r="5"
    style="fill: red; stroke:black; stroke-width: 2;"/>
```

而是指可以编写每个表现属性为独立的属性：

```
<circle cx="10" cy="10" r="5"
    fill="red" stroke="black" stroke-width="2"/>
```

如果你觉得这样做混合了结构与表现，那么你是对的。不过，当我们通过将 XML 数据源转换为 SVG 的方式来创建 SVG 文档时，表现属性会派上用场，我们将在第 15 章看到这样的情况。在这些情况下，为每个表现属性创建独立的属性，要比为一个 `style` 属性创建内容更容易。如果使用 SVG 的环境不支持样式表，可能需要使用表现属性。

表现属性位于优先级列表的最底部。任何来自内联样式、内部样式表或者外部样式表的样式声明都会覆盖表现属性，但表现属性会覆盖继承的样式。在下面的 SVG 文档中，圆会被填充为红色，而不是绿色：

```
<svg width="200" height="200"
  xmlns="http://www.w3.org/2000/svg">
  <defs>
    <style type="text/css"><![CDATA[
      circle { fill: red; }
    ]]></style>
  </defs>
  <circle cx="20" cy="20" r="15" fill="green"/>
</svg>
```

我们再次强调，指定表现信息的首选应该是使用 `style` 属性或者样式表。样式表允许我们为文档中的某些元素应用一系列复杂的填充和笔画特性，而不必为每个元素复制一遍表现信息，这正是表现属性所应该有的效果。样式表的能力和灵活性允许我们花最少的精力就能改变对多个文档的外观。

5.3 分组和引用对象

虽然可以将所有的绘图看成是由一系列几乎一样的形状和线条组成的，但通常我们还是认为大多数非抽象的艺术作品都是由一系列命名对象组成的，而这些对象由形状和线条组合而成。SVG 提供了一些元素，允许我们对元素进行这样的分组，从而使文档更加结构化、更易理解。

5.3.1 <g>元素

`<g>` 元素会将其所有子元素作为一个组合，通常组合还会有一个唯一的 `id` 作为名称。每个组合还可以拥有自己的 `<title>` 和 `<desc>` 来供基于文本的 XML 应用程序识别，或者为视障用户提供更好的可访问性。许多 SVG 渲染代理都会在鼠标悬停或者轻触组合内的图形时显示一个提示框，显示 `<title>` 元素的内容。屏幕阅读器也会读取 `<title>` 和 `<desc>` 元素的内容。

`<g>` 元素可以组合元素，并为它们提供一些注解，这使得我们的文档结构更为清晰。除此之外，`<g>` 元素还提供了一些书写上的便利。在起始 `<g>` 标签中指定的所有样式会应用于组合内的所有子元素。如示例 5-5，我们可以不用复制如图 5-4 所示的每个元素上的 `style="fill:none; stroke:black;"`。而且组合还可以彼此嵌套，我们会在第 6 章看到这样的例子。

`<g>` 元素类似于 Adobe Illustrator 等程序中的组合对象（Group Objects）功能。它还提供了一个功能，类似于这些程序中"层"的概念，一个层就是一些相关对象构成的分组。

示例 5-5：简单使用 `<g>` 元素

```
<svg width="240px" height="240px" viewBox="0 0 240 240"
  xmlns="http://www.w3.org/2000/svg">
<title>Grouped Drawing</title>
<desc>Stick-figure drawings of a house and people</desc>

<g id="house" style="fill: none; stroke: black;">
  <desc>House with door</desc>
  <rect x="6" y="50" width="60" height="60"/>
  <polyline points="6 50, 36 9, 66 50"/>
  <polyline points="36 110, 36 80, 50 80, 50 110"/>
</g>

<g id="man" style="fill: none; stroke: black;">
  <desc>Male human</desc>
  <circle cx="85" cy="56" r="10"/>
  <line x1="85" y1="66" x2="85" y2="80"/>
  <polyline points="76 104, 85 80, 94 104" />
  <polyline points="76 70, 85 76, 94 70" />
</g>
```

```
<g id="woman" style="fill: none; stroke: black;">
  <desc>Female human</desc>
  <circle cx="110" cy="56" r="10"/>
  <polyline points="110 66, 110 80, 100 90, 120 90, 110 80"/>
  <line x1="104" y1="104" x2="108" y2="90"/>
  <line x1="112" y1="90" x2="116" y2="104"/>
  <polyline points="101 70, 110 76, 119 80"/>
</g>
</svg>
```

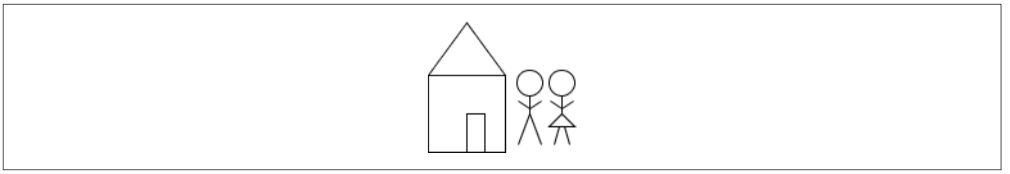


图 5-4：分组火柴人

5.3.2 <use>元素

复杂的图形中经常会出现重复的元素。比如，产品手册可能在每一页的左上角和右下角都有公司的标志。如果我们使用绘图设计程序绘制宣传册，只需绘制一次标志，然后将所有的元素组合在一起，就能将它们复制并粘贴到其他位置。SVG 使用 `<use>` 元素，为定义在 `<g>` 元素内的组合或者任意独立图形元素（比如只想定义一次的复杂多边形形状）提供了类似复制粘贴的能力。

定义了一组图形对象后，可以使用 `<use>` 标签再次显示它们。要指定想要重用的组合，给 `xlink:href` 属性指定 URI 即可，同时还要指定 `x` 和 `y` 的位置以表示组合的 (0, 0) 应该移动到的位置（6.1 节会介绍实现这一效果的另一种方法）。因此，为了创建另一个如图 5-5 那样的房子和一组小人，只要把这些线条放到闭合 `</svg>` 标签之前即可。

```
<use xlink:href="#house" x="70" y="100"/>
<use xlink:href="#woman" x="-80" y="100"/>
<use xlink:href="#man" x="-30" y="100"/>
```

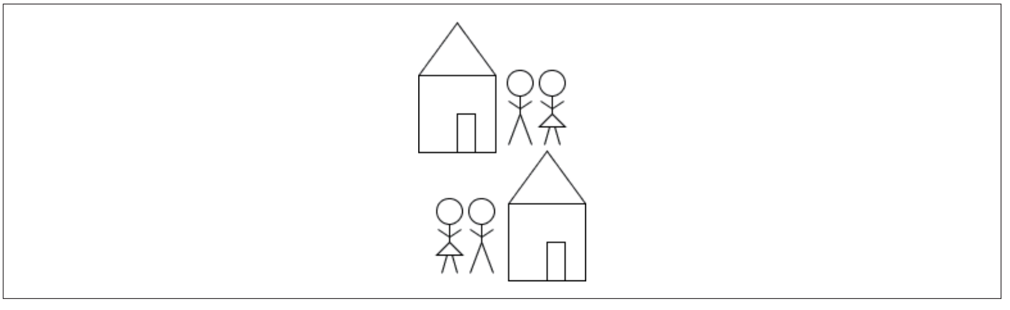


图 5-5：复用组合的火柴人

5.3.3 <defs>元素

你可能已经注意到前面那个例子有以下几个缺点。

- 复用 man 和 woman 组合时，需要知道原始图像中这些图形的位置，并以此位置作为复用的基础，而不是使用诸如 0 这样简单的数字。
- 房子的填充和笔画颜色由原始图形建立，并且不能通过 <use> 元素覆盖。这意味着我们不能构造一行彩色的房子。
- 文档中会画出所有的三个元素 woman、man 和 house。我们并不能将它们单独“存储”下来，然后只绘制一排房子或者只绘制一组人。

<defs>（定义）元素可以解决这些问题。通过在起始和结束 <defs> 标记之间放置这些组合对象，我们可以告诉 SVG 只定义但不显示它们。实际上，SVG 规范推荐我们将所有想要复用的对象放置在 <defs> 元素内，这样 SVG 阅读器进入流式环境中就能更高效地处理数据。在示例 5-6 中，house、man 和 woman 被定义在左上角 (0, 0) 处，并且没有为房子指定任何填充颜色。由于组合在 <defs> 元素内，它们不会立刻绘制到屏幕上，而是作为“模板”供其他地方使用。我们还构建了另一个命名为 couple 的组合，它通过 <use> 使用 man 和 woman 组合。（注意图 5-6 中下半部分并不能使用 couple，因为图形的排列不一样。）

示例 5-6：<defs> 元素

http://oreillymedia.github.io/svg-essentials-examples/ch05/defs-example.html

```
<svg width="240px" height="240px" viewBox="0 0 240 240"
  xmlns="http://www.w3.org/2000/svg">
<title>Grouped Drawing</title>
<desc>Stick-figure drawings of a house and people</desc>

<defs>
<g id="house" style="stroke: black;">
  <desc>House with door</desc>
  <rect x="0" y="41" width="60" height="60"/>
  <polyline points="0 41, 30 0, 60 41"/>
  <polyline points="30 101, 30 71, 44 71, 44 101"/>
</g>

<g id="man" style="fill: none; stroke: black;">
  <desc>Male stick figure</desc>
  <circle cx="10" cy="10" r="10"/>
  <line x1="10" y1="20" x2="10" y2="44"/>
  <polyline points="1 58, 10 44, 19 58"/>
  <polyline points="1 24, 10 30, 19 24"/>
</g>

<g id="woman" style="fill: none; stroke: black;">
  <desc>Female stick figure</desc>
  <circle cx="10" cy="10" r="10"/>
  <polyline points="10 20, 10 34, 0 44, 20 44, 10 34"/>
```

```
    <line x1="4" y1="58" x2="8" y2="44"/>
    <line x1="12" y1="44" x2="16" y2="58"/>
    <polyline points="1 24, 10 30, 19 24" />
  </g>

  <g id="couple">
    <desc>Male and female stick figures</desc>
    <use xlink:href="#man" x="0" y="0"/>
    <use xlink:href="#woman" x="25" y="0"/>
  </g>
  </defs>

  <!-- 利用组合定义 -->
  <use xlink:href="#house" x="0" y="0" style="fill: #cfc;"/>
  <use xlink:href="#couple" x="70" y="40"/>

  <use xlink:href="#house" x="120" y="0" style="fill: #99f;"/>
  <use xlink:href="#couple" x="190" y="40"/>

  <use xlink:href="#woman" x="0" y="145"/>
  <use xlink:href="#man" x="25" y="145"/>
  <use xlink:href="#house" x="65" y="105" style="fill: #c00;"/>
  </svg>
```

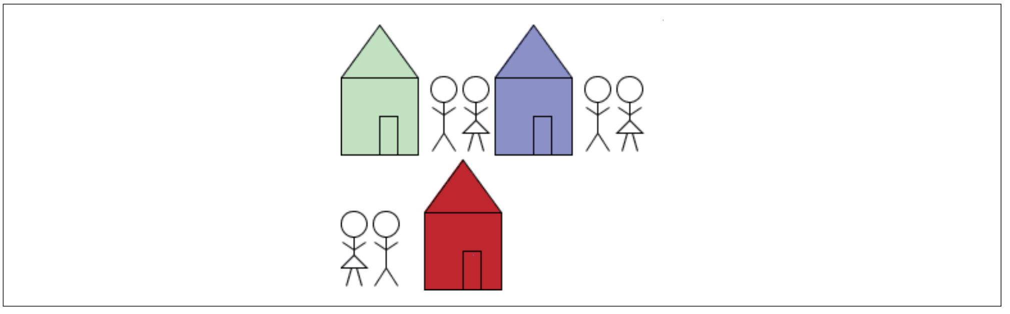


图 5-6：在 <defs> 内使用组合的结果

<use> 元素并不限制只能使用同一文件内的对象，事实上 xlink:href 属性可以指定任意有效的文件或者 URI。这使得我们可以将一组公用元素集合在一个 SVG 文件内，然后在其他文件中选择性地使用它们。比如，我们可以创建一个名为 identity.svg 的文件，该文件包含你的组织要使用的所有标识图形：

```
  <g id="company_mascot">
     <!-- 绘制企业吉祥物 -->
  </g>

  <g id="company_logo" style="stroke: none;">
     <polygon points="0 20, 20 0, 40 20, 20 40"
        style="fill: #696;"/>
     <rect x="7" y="7" width="26" height="26"
         style="fill: #c9c;"/>
```

```
</g>

<g id="partner_logo">
    <!-- 绘制合作伙伴的Logo -->
</g>
```

然后使用如下方式引用它们：

```
<use xlink:href="identity.svg#company_logo" x="200" y="200"/>
```

出于安全的原因，并非所有的 SVG 阅读器都支持外部引用，尤其是 Web 浏览器。有些浏览器（尤其是 IE）完全不支持外部文件引用。其他浏览器也只允许 `<use>` 元素引用同一域下的文件或者专门配置了允许跨域使用的 Web 服务器上的文件。

5.3.4 <symbol>元素

`<symbol>` 元素提供了另一种组合元素的方式。和 `<g>` 元素不同，`<symbol>` 元素永远不会显示，因此我们无需把它放在 `<defs>` 规范内。然而，我们仍然习惯将它放到 `<defs>` 中，因为 symbol 也是我们定义的供后续使用的元素。symbol 还可以指定 `viewBox` 和 `preserveAspectRatio` 属性，通过给 `<use>` 元素添加 `width` 和 `height` 属性就可以让 symbol 适配视口大小。示例 5-7 展示了忽略 `width` 和 `height` 的简单组合（前两个八边形），但在使用 symbol 时使用了这两个属性。图 5-7 右下角的八边形边缘被裁减了，这是因为 `preserveAspectRatio` 属性被设置为 `slice`。其包含的 `<rect>` 元素展示了每个 `<use>` 元素的坐标。

邮
电

示例 5-7：symbol 与组合

http://oreillymedia.github.io/svg-essentials-examples/ch05/symbol.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
<title>Symbols vs. groups</title>
<desc>Use</desc>

<defs>
  <g id="octagon" style="stroke: black;">
    <desc>Octagon as group</desc>
    <polygon points="
      36 25, 25 36, 11 36, 0 25,
      0 11, 11 0, 25 0, 36 11"/>
  </g>

  <symbol id="sym-octagon" style="stroke: black;"
    preserveAspectRatio="xMidYMid slice" viewBox="0 0 40 40">
    <desc>Octagon as symbol</desc>
    <polygon points="
```

```
      36 25, 25 36, 11 36, 0 25,
      0 11, 11 0, 25 0, 36 11"/>
  </symbol>
</defs>

<g style="fill:none; stroke:gray">
  <rect x="40" y="40" width="30" height="30"/>
  <rect x="80" y="40" width="40" height="60"/>
  <rect x="40" y="110" width="30" height="30"/>
  <rect x="80" y="110" width="40" height="60"/>
</g>
<use xlink:href="#octagon" x="40" y="40" width="30" height="30"
  style="fill: #c00;"/>
<use xlink:href="#octagon" x="80" y="40" width="40" height="60"
  style="fill: #cc0;"/>
<use xlink:href="#sym-octagon" x="40" y="110" width="30" height="30"
  style="fill: #cfc;"/>
<use xlink:href="#sym-octagon" x="80" y="110" width="40" height="60"
  style="fill: #699;"/>
</svg>
```

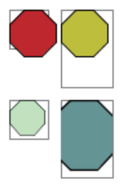


图 5-7：组合与 symbol

5.3.5 <image>元素

<use> 元素允许我们复用 SVG 文件的一部分，而 <image> 元素可以包含一个完整的 SVG 或者栅格文件。如果包含一个 SVG 文件，其视口会基于引用文件的 x、y、width 和 height 属性来建立；如果包含一个栅格文件，它会被缩放以适配该属性指定的矩形。SVG 规范要求阅读器支持 JPEG 和 PNG 两种栅格文件，阅读器还可能支持其他文件，比如大多数浏览器还支持 GIF。示例 5-8 展示了如何在 SVG 文件内包含 JPEG 文件。结果如图 5-8 所示。

示例 5-8：使用 <defs> 元素

```
<svg width="310px" height="310px" viewBox="0 0 310 310"
  xmlns="http://www.w3.org/2000/svg">

<ellipse cx="154" cy="154" rx="150" ry="120" style="fill: #999999;"> ❶
<ellipse cx="152" cy="152" rx="150" ry="120" style="fill: #cceeff;"> ❷

<image xlink:href="kwanghwamun.jpg"  ❸
  x="72" y="92"     ❹
  width="160" height="120"/>     ❺

</svg>
```

❶ 创建一个灰色椭圆模拟投影。[1]

❷ 创建主蓝色椭圆形。因为它在灰色椭圆之后出现，因此它显示在灰色椭圆之上。

❸ 指定要包含文件的 URI。

❹ 指定图像的左上角位置。

❺ 指定图片应该被缩放的宽度和高度。



图 5-8：包含在 SVG 文件内的 JPEG 图像

如果图像文件的尺寸与元素的宽度和高度不匹配，`<image>` 元素可以使用 `preserveAspectRatio` 属性指示浏览器应该怎么处理。其默认值为 `xMidYMid meet`，这会缩放图像并居中显示在指定的矩形中（可查看 3.4 节）。如果包含一个 SVG 文件，还可以在 `preserveAspectRatio` 值的开头添加 `defer` 关键字（比如 `defer xMidTMid meet`）；这样如果包含的图像也有 `preserveAspectRatio` 属性，则会使用图像的属性来替代默认值。

注 1：11.2 节我们会看到另一种创建投影的方式。

第 6 章

坐标系统变换

到目前为止，所有图形都是什么样就显示成什么样，即按照属性中指定的位置和方式来显示。有时候我们可能想要旋转、缩放或者移动图片到新的位置。为了完成这些任务，我们可以给对应的 SVG 元素添加 transform 属性。本章会研究这些变换的细节。

6.1 translate变换

在第 5 章我们已经见过，可以为 <use> 元素使用 x 和 y 属性，以在特定的位置放置图形对象组合。查看示例 6-1 中的 SVG，它定义了一个正方形并将它绘制在网格的左上角，然后在左上角坐标 (50, 50) 处重新绘制了它。图 6-1 中的虚线并不是 SVG 的一部分，它只是用来展示画布。

示例 6-1：使用 <use> 移动图形

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
    <g id="square">
        <rect x="0" y="0" width="20" height="20"
            style="fill: black; stroke-width: 2;"/>
    </g>
    <use xlink:href="#square" x="50" y="50"/>
</svg>
```

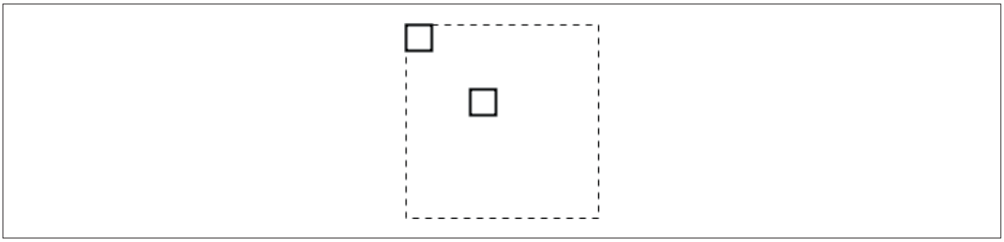


图 6-1：使用 <use> 移动正方形

事实上，x 和 y 值只是更常用和更强大的 transform 属性的一种简写形式。具体来说，x 和 y 相当于属性 transform="translate(x-value, y-value)"，而 translate 只是一个花哨的表示“移动”的术语。x 和 y 根据当前用户坐标系统计算。让我们使用 transform 生成另一个正方形，它的左上角位于 (50, 50) 处。示例 6-2 列出了 SVG 代码。

示例 6-2：用 translate 移动坐标系统

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
    <g id="square">
        <rect x="0" y="0" width="20" height="20"
            style="fill: none; stroke:black; stroke-width: 2;"/>
    </g>
    <use xlink:href="#square" transform="translate(50,50)"/>
</svg>
```

结果看起来和图 6-1 完全一样。你可能认为它是像图 6-2 那样通过移动正方形到网格的另一个地方完成的，但其实并不是这样。

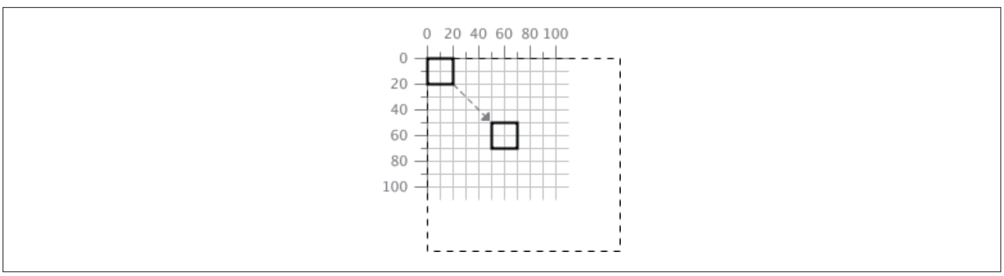


图 6-2：移动看起来是如何工作的（但实际上不是）

实际上完全不同。translate 声明会获取整个网格，然后把它移动到画布的新位置，而不是移动正方形。就正方形而言，它仍然绘制在左上角 (0, 0) 处，如图 6-3 所示。

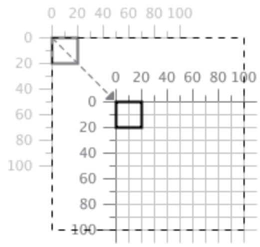


图 6-3：`translate` 的移动原理

在线示例允许我们尝试不同的坐标：

http://oreillymedia.github.io/svg-essentials-examples/ch06/translate.html



变换永远不会改变图形对象的网格坐标，但是它会改变网格在画布上的位置。

乍看之下，使用 `translate` 似乎荒谬且低效，如同通过移动整个起居室、墙壁以及所有东西到新的位置，从而让沙发远离房子的外墙。事实上，如果 `translation` 是唯一可用的变换，那么移动整个坐标系统将是一种浪费。但是，我们很快会看到，如果要对整个坐标系统应用其他的变换或者一系列变换的组合，那么从数学和概念上讲，这样做会更方便。

6.2 scale变换

它可以通过缩放坐标系统的方式让对象显示得比定义的尺寸更大或者更小。这种变换被指定为如下形式：

- `transform="scale(`*`value`*`)"`

  所有的 *x* 和 *y* 坐标乘以给定的 `value`。

- `transform="scale(`*`x-value, y-value`*`)"`

  所有 *x* 坐标乘以给定的 `x-value`，所有 *y* 坐标乘以给定的 `y-value`。

示例 6-3 是第一类缩放变换的例子，均匀地将两个轴放大一倍。再次声明，图 6-4 中的虚线并非 SVG 的一部分，它们只是显示我们所感兴趣的画布区域。注意正方形的左上角在 (10, 10) 位置。

示例 6-3：均匀地缩放图形

http://oreillymedia.github.io/svg-essentials-examples/ch06/scale.html

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
    <g id="square">
        <rect x="10" y="10" width="20" height="20"
            style="fill: none; stroke: black;"/>
    </g>
    <use xlink:href="#square" transform="scale(2)"/>
</svg>
```

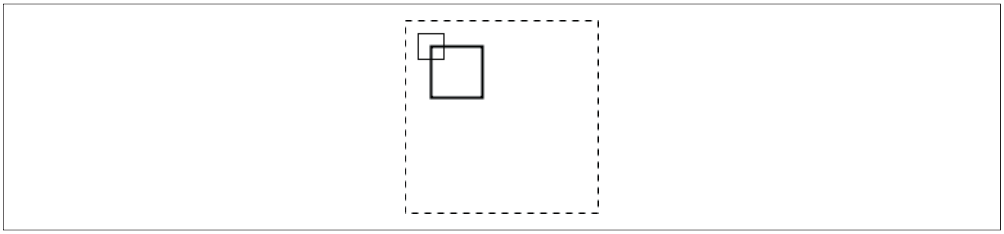


图 6-4：缩放变换的结果

你可能会想：“等等，我可以理解为什么正方形变大了。但是这里并没使用 translate，为什么正方形在不同的位置了呢？”让我们看看图 6-5 到底发生了什么，一切就会变得清晰。网格并没有被移动，坐标系统的点 (0, 0) 仍然在相同的位置，但是每个用户坐标都变成原来的两倍了。从网格线上可以看到，矩形的左上角在更大的新网格中仍然在 (10, 10) 位置，因为对象并没有移动。这也解释了为什么较大正方形的轮廓线更粗了。stroke-width 仍然是一个用户单位，但是这个单位已经是原来的两倍了，因此其笔画变粗了。

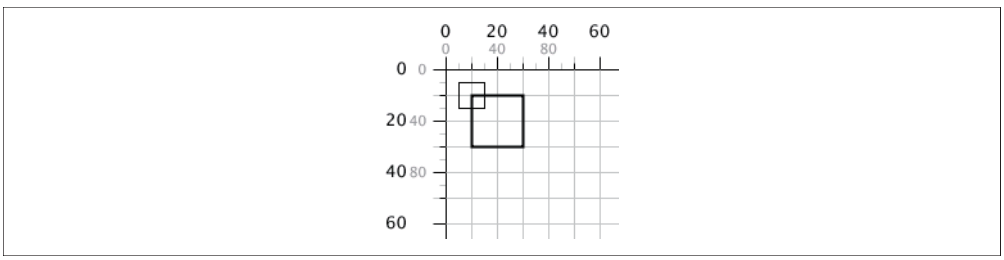


图 6-5：scale 变换的工作原理

> 缩放变换永远不会改变图形对象的网格坐标或者它的笔画宽度，但是，它会改变对应画布上的坐标系统（网格）的大小。

我们可以通过使用 scale 变换的第二种形式，为坐标系统的 *x* 轴和 *y* 轴指定不同的缩放比

例。示例 6-4 绘制的正方形的 x 轴被放大了 3 倍，y 轴被放大了 1.5 倍。正如在图 6-6 中可以看到的，一个单位的笔画宽度也被不均匀地缩放了。

示例 6-4：不均匀地缩放图形

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <g id="square">
    <rect x="10" y="10" width="20" height="20"
      style="fill: none; stroke: black;"/>
  </g>
  <use xlink:href="#square" transform="scale(3, 1.5)"/>
</svg>
```

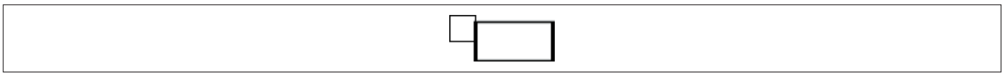


图 6-6：不均匀地缩放变换的结果

到目前为止，所有示例都只对 <use> 元素应用了 transform 属性。我们可以将一系列元素组合在一起，然后对组合应用变换：

```
<g id="group1" transform="translate(3, 5)">
  <line x1="10" y1="10" x2="30" y2="30"/>
  <circle cx="20" cy="20" r="10"/>
</g>
```

我们也可以为单个对象或者基本形状应用变换。比如，这里有一个矩形，其坐标系统被放大了 3 倍：

```
<rect x="15" y="20" width="10" height="5"
  transform="scale(3)"
  style="fill: none; stroke: black;"/>
```

很明显，缩放后矩形的宽度和高度应该是未缩放矩形的 3 倍。然而，你可能想知道 x 坐标和 y 坐标的值是在矩形被缩放前还是缩放后计算的。答案是 SVG 会在计算形状的坐标之前，先对坐标系统应用变换。示例 6-5 是被缩放的矩形的 SVG，图 6-7 上的网格线是在坐标系统缩放前绘制的。

示例 6-5：单个图形对象的变换

```
<!-- 未缩放坐标系统的网格参考线 -->
<line x1="0" y1="0" x2="100" y2="0" style="stroke: black;"/>
<line x1="0" y1="0" x2="0" y2="100" style="stroke: black;"/>
<line x1="45" y1="0" x2="45" y2="100" style="stroke: gray;"/>
<line x1="0" y1="60" x2="100" y2="60" style="stroke: gray;"/>

<!-- 要变换的矩形 -->
<rect x="15" y="20" width="10" height="5"
    transform="scale(3)"
    style="fill: none; stroke: black;"/>
```

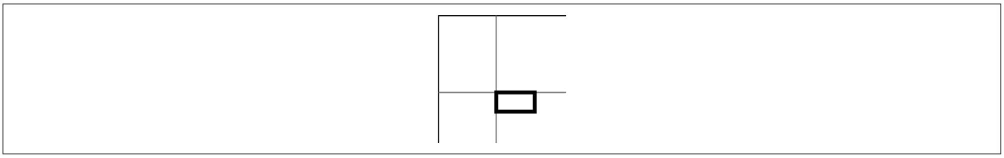


图 6-7：单个图形变换的结果

为形状应用变换与形状被包含在变换组合中的效果相同。在前面的例子中，被缩放的矩形等价于这个 SVG：

```
<g transform="scale(3)">
  <rect x="15" y="20" width="10" height="5"
    style="fill: none; stroke: black;"/>
</g>
```

6.3　变换序列

一个图形对象上可以做多个变换。我们只需将多个变换通过空格或逗号分隔，依次放入 transform 属性即可。下面是一矩形，它经过了两个变换，先平移再缩放（通过绘制的轴可以看出来矩形确实被移动了）。

```
<!-- 绘制轴 -->
<line x1="0" y1="0" x2="0" y2="100" style="stroke: gray;"/>
<line x1="0" y1="0" x2="100" y2="0" style="stroke: gray;"/>

<rect x="10" y="10" height="15" width="20"
  transform="translate(30, 20) scale(2)"
  style="fill: gray;"/>
```

这个例子等价于下面的嵌套组合序列，这两个例子都会生成如图 6-8 所示的结果。

```
<g transform="translate(30, 20)">
  <g transform="scale(2)">
  <rect x="10" y="10" height="15" width="20"
    style="fill: gray;"/>
  </g>
</g>
```

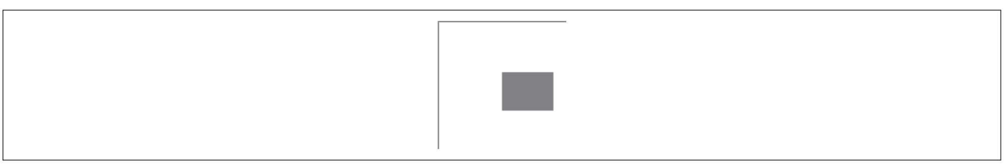


图 6-8：先平移后缩放的结果

图 6-9 展示了每个变换阶段发生了什么。

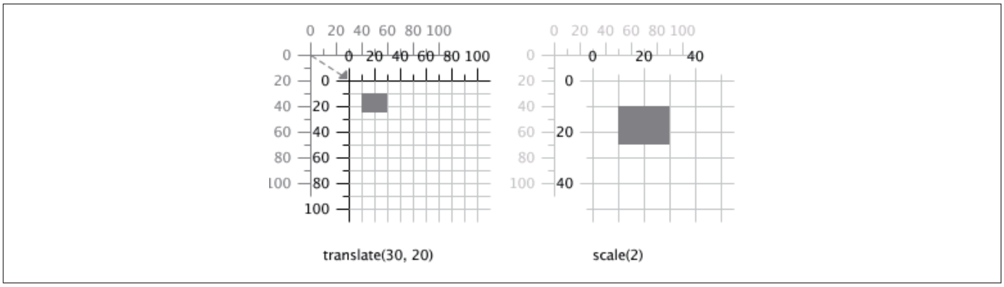


图 6-9：先平移后缩放的工作原理



变换序列的顺序会影响结果。一般情况下，变换 A 然后变换 B 的结果与变换 B 然后变换 A 的结果不同。

示例 6-6 绘制了与前面例子相同的灰色矩形。然后又绘制了一个黑色矩形，但是这次 `scale` 在 `translate` 之前。从图 6-10 所示的结果中可以看到，矩形最终在画布的不同位置。

示例 6-6：变换序列——先缩放后平移

```
<!-- 绘制轴 -->
<line x1="0" y1="0" x2="0" y2="100" style="stroke: gray;"/>
<line x1="0" y1="0" x2="100" y2="0" style="stroke: gray;"/>

<rect x="10" y="10" width="20" height="15"
    transform="translate(30, 20) scale(2)" style="fill: gray;"/>

<rect x="10" y="10" width="20" height="15"
    transform="scale(2) translate(30, 20)"
    style="fill: black;"/>
```

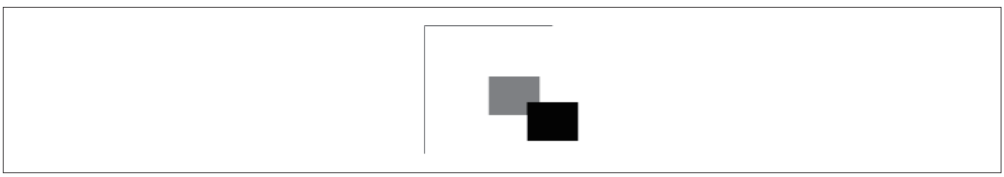


图 6-10：先缩放后平移的结果

黑色矩形最终远离原始矩形是因为首先应用了缩放，因此横向平移 20 单位和纵向平移 10 单位是在单位放大一倍之后完成的，如图 6-11 所示。

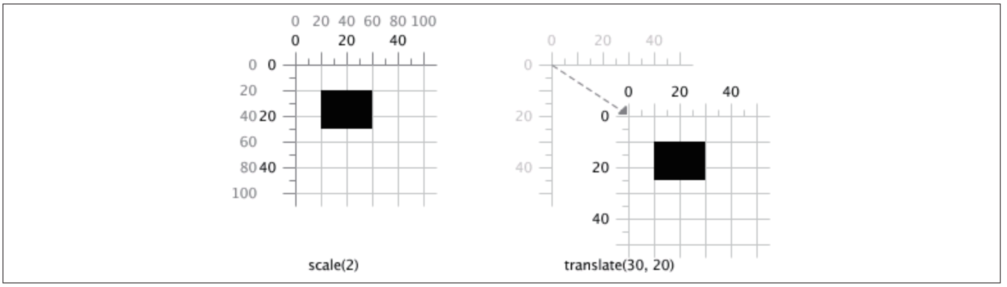


图 6-11：先缩放后平移的工作原理

在在线示例中，我们可以尝试任意的变换序列，然后比较原始矩形和变换后的矩形：

http://oreillymedia.github.io/svg-essentials-examples/ch06/sequence.html

6.4 技巧：笛卡儿坐标系统转换

如果从其他系统传输数据到 SVG，你可能必须处理使用笛卡儿坐标（在高中代数中学到的）表示数据的矢量图形。在这个系统中，点 (0, 0) 位于画布的左下角，*y* 坐标向上递增。图 6-12 展示了使用笛卡儿坐标绘制的梯形的坐标。

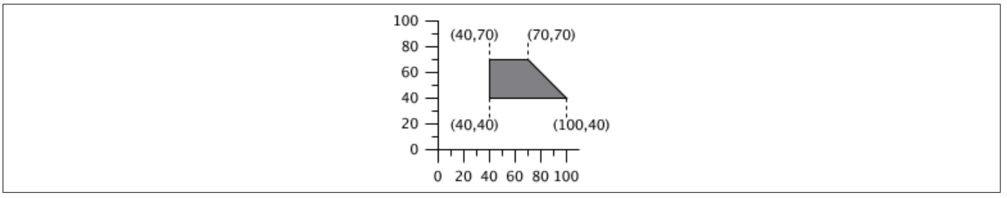


图 6-12：使用笛卡儿坐标绘制的梯形

y 轴与 SVG 的默认约定“上下相反”，因此需要重新计算坐标。我们可以使用变换序列让 SVG 做这些工作，而不是手动处理。首先，平移图像到 SVG 中，其坐标如示例 6-7 所示（这个例子还包括轴作为参考）。如我们所料，图像会上下翻转。注意，图 6-13 并不是从左向右翻转，因为 *x* 轴的方向在笛卡儿坐标和默认 SVG 坐标系统中的方向相同。

示例 6-7：直接使用笛卡儿坐标

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <!-- 轴 -->
  <line x1="0" y1="0" x2="100" y2="0" style="stroke: black;"/>
  <line x1="0" y1="0" x2="0" y2="100" style="stroke: black;"/>

  <!-- 梯形 -->
  <polygon points="40 40, 100 40, 70 70, 40 70"
      style="fill: gray; stroke: black;"/>
</svg>
```

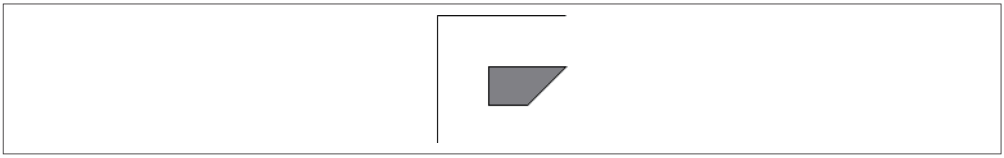


图 6-13：使用原始笛卡儿坐标的结果

要把图片翻转回正面朝上，我们可以利用这一事实：通过负值缩放形状会反转坐标顺序。然而，由于整个网格最终会翻转到坐标 0 点的另一侧，我们还需要将形状平移回画布的可视部分。这一转换遵循以下步骤。

（1）在原始绘图中找到最大 y 坐标。在本例中是 100，也就是原始 y 轴的末端。

（2）将整个绘图放入 <g> 元素中。

（3）启用平移，根据最大 y 值向下移动坐标系统：`transform=”translate(0, max-y)”`。

（4）接下来的变换就是缩放 y 轴 −1 倍，让它倒置翻转：`transform=”translate(0, max-y) scale(1, -1)”`。

我们并不希望改变 x 轴的值，但是仍然需要为 `translate` 和 `scale` 函数指定一个 x 值。什么都不做的平移，指定 0 即可，但是什么都不做的缩放，要变换值为 1，因为坐标会乘以缩放比例。`scale(0)` 变换会将形状折叠为一个点（因为每个坐标乘以 0 将变成 0）。

示例 6-8 使用了这种变换，生成了如图 6-14 所示的正面朝上的梯形。

示例 6-8：笛卡儿坐标变换

```
<svg width="200px" height="200px" viewBox="0 0 200 200"
  xmlns="http://www.w3.org/2000/svg">
  <g transform="translate(0,100) scale(1,-1)">
    <!-- 轴 -->
    <line x1="0" y1="0" x2="100" y2="0" style="stroke: black;"/>
    <line x1="0" y1="0" x2="0" y2="100" style="stroke: black;"/>

    <!-- 梯形 -->
    <polygon points="40 40, 100 40, 70 70, 40 70"
        style="fill: gray; stroke: black;"/>
  </g>
</svg>
```

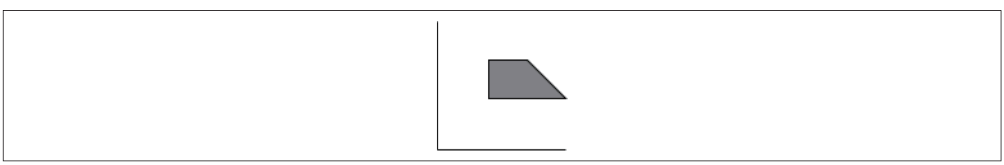


图 6-14：笛卡儿坐标变换

6.5 rotate变换

还可以根据指定的角度旋转坐标系统。在默认的坐标系统中，角度的测量是按顺时针增加的，水平线的角度为 0 度，如图 6-15 所示。

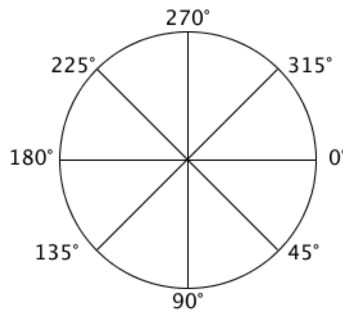


图 6-15：默认角度测量

除非我们另行指定，否则旋转的中心点（轴心点的别称）被假定为 (0, 0)。示例 6-9 绘制了一个灰色的正方形，然后在坐标系统旋转 45 度后又绘制了一个的黑色正方形。轴仍然用作参考。图 6-16 展示了结果。如果你发现正方形的位置似乎也移动了，请不要感到惊讶。记住，如图 6-17 所示，被旋转的是整个坐标系统。[1]

示例 6-9：围绕原点旋转

http://oreillymedia.github.io/svg-essentials-examples/ch06/rotate.html

```
<!-- 轴 -->
<polyline points="100 0, 0 0, 0 100" style="stroke: black; fill: none;"/>

<!-- 默认和旋转后的正方形 -->
<rect x="70" y="30" width="20" height="20" style="fill: gray;"/>
<rect x="70" y="30" width="20" height="20"
    transform="rotate(45)" style="fill: black;"/>
```

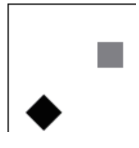


图 6-16：围绕原点旋转的结果

注 1：本章所有示意图都是静态图片。这幅图显示了两个正方形（一个旋转过，另一个没有）。展示旋转正方形动画要使用，我们将会在 12.9 节讨论。

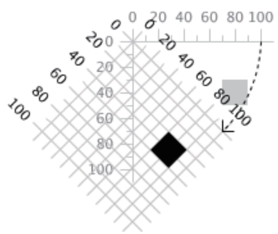


图 6-17：围绕原点旋转的工作原理

很多时候，我们并不想围绕原点旋转整个坐标系统，而是希望围绕某个点旋转单个对象。我们可以通过一系列变换做到这一点：`translate(centerX, centerY) rotate(angle) translate(-centerX, -centerY)`。SVG 提供了另一个版本的 `rotate`，让处理这一常见任务更容易。在 `rotate` 变换的第二种形式中，指定角度以及想要围绕其旋转的中心点即可。

```
rotate(angle, centerX, centerY)
```

这样做的效果是以指定的 *x* 和 *y* 点作为原点临时建立一个新的坐标系统执行旋转操作，然后重新建立原始坐标。示例 6-10 展示了这种形式的旋转（`rotate`）创建了箭头的多个副本，如图 6-18 所示。

示例 6-10：围绕中心点旋转

http://oreillymedia.github.io/svg-essentials-examples/ch06/rotate-center.html

```
<!-- 旋转中心 -->
<circle cx="50" cy="50" r="3" style="fill: black;"/>

<!-- 未旋转的箭头 -->
<g id="arrow" style="stroke: black;">
    <line x1="60" y1="50" x2="90" y2="50"/>
    <polygon points="90 50, 85 45, 85 55"/>
</g>

<!-- 围绕中心点旋转 -->
<use xlink:href="#arrow" transform="rotate(60, 50, 50)"/>
<use xlink:href="#arrow" transform="rotate(-90, 50, 50)"/>
<use xlink:href="#arrow" transform="rotate(-150, 50 50)"/>
```

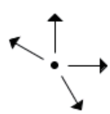


图 6-18：围绕中心点旋转的结果

6.6 技巧：围绕中心点缩放

虽然可以围绕某个不是原点的点旋转，但是却不能围绕某个点缩放（scale）。然而，我们可以使用一系列简单的变换制造同心符号。要围绕某个点按照给定的比例缩放对象可以这么做：

```
translate(-centerX*(factor-1), -centerY*(factor-1))
scale(factor)
```

你可能还希望将 stroke-width 的值也除以缩放系数，从而让变大后的对象的轮廓保持同样的宽度。示例 6-11 绘制了一系列同心矩形，如图 6-19 所示。[2]

示例 6-11：围绕中心点缩放

```
<!-- 缩放中心 -->
<circle cx="50" cy="50" r="2" style="fill: black;"/>

<!-- 未缩放的矩形 -->
<g id="box" style="stroke: black; fill: none;">
    <rect x="35" y="40" width="30" height="20"/>
</g>

<use xlink:href="#box" transform="translate(-50, -50) scale(2)"
    style="stroke-width: 0.5;"/>
<use xlink:href="#box" transform="translate(-75, -75) scale(2.5)"
    style="stroke-width: 0.4;"/>
<use xlink:href="#box" transform="translate(-100, -100) scale(3)"
    style="stroke-width: 0.33;"/>
```

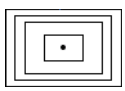


图 6-19：围绕中心点缩放的结果

6.7 skewX和skewY变换

SVG 还有另外两种变换：skewX 和 skewY，这让我们可以倾斜某个轴。其一般形式为 skewX(angle) 和 skewY(angle)。skewX 变换会按照指定的角度“推动”所有 x 坐标，y 坐标不会改变。skewY 会倾斜 y 坐标，而 x 坐标不会改变，如图 6-20，由示例 6-12 中的代码绘制。

注 2：这也是静态图片，是一个“方形靶心”。如果想要展示扩展正方形的动画，要使用，我们将会在 12.9 节讨论。

示例 6-12：skewX 和 skewY

http://oreillymedia.github.io/svg-essentials-examples/ch06/skew.html

```
<!-- 参考线 -->  ❶
<g style="stroke: gray; stroke-dasharray: 4 4;">
    <line x1="0" y1="0" x2="200" y2="0"/>
    <line x1="20" y1="0" x2="20" y2="90"/>
    <line x1="120" y1="0" x2="120" y2="90"/>
</g>

<g transform="translate(20, 0)">     ❷
    <g transform="skewX(30)">     ❸
        <polyline points="50 0, 0 0, 0 50"      ❹
             style="fill: none; stroke: black; stroke-width: 2;"/>
        <text x="0" y="60">skewX</text>     ❺
    </g>
</g>

<g transform="translate(120, 0)">     ❻
    <g transform="skewX(30)">
        <polyline points="50 0, 0 0, 0 50"
            style="fill: none; stroke: black; stroke-width: 2;"/>
        <text x="0" y="60">skewX</text>
    </g>
</g>
```

❶ 执行变换前这些虚线绘制在默认坐标系统中。

❷ 将整个倾斜“打包”移动到希望的位置。

❸ 倾斜 *x* 坐标 30 度。这一变换不会改变原点，新坐标系统中原点仍然在 (0, 0) 处。

❹ 方便起见，在原点绘制对象。

❺ 第 9 章会详细介绍文本。

❻ 这些元素的组织方式与前面的元素一样，只是 *y* 坐标被倾斜了。

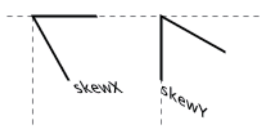


图 6-20：skewX 和 skewY 变换的结果

注意，skewX 保持水平线不变，skewY 则保持垂直线不动。想想为什么吧。

6.8　变换总结

表 6-1 总结了可用于 SVG 中的变换。

表6-1：SVG变换

变　换	描　述
`translate(x, y)`	按照指定的 x 和 y 值移动用户坐标系统。注意：如果没有指定 y 值，则假定为 0
`scale(xFactor, yFactor)`	使用指定的 `xFactor` 和 `yFactor` 乘以所有的用户坐标系统。比例值可以是小数或者负值
`scale(factor)`	和 `scale(factor, factor)` 相同
`rotate(angle)`	按照指定的 `angle` 旋转用户坐标。旋转中心为原点 (0, 0)。在默认坐标系统中，旋转角度按顺时针方向递增，水平线的角度为 0 度
`rotate(angle, centerX, centerY)`	按照指定的 `angle` 旋转用户坐标。旋转中心由 `centerX` 和 `centerY` 指定
`skewX(angle)`	根据指定的 `angle` 倾斜所有 x 坐标。从视觉上讲，这会让垂直线出现角度
`skewY(angle)`	根据指定的 `angle` 倾斜所有 y 坐标。从视觉上讲，这会让水平线出现角度
`matrix(a b c d e f)`	指定一个六个值组成的矩阵变换，参考附录 D

6.9　CSS变换和SVG

在编写本文时，CSS 变换模块还处在草案阶段（http://www.w3.org/TR/css-transforms-1/）。由于它还是一个草案，所以其细节可能还会改变，并且浏览器的支持情况也可能不同。如果你已经在使用 CSS 变换了，以下是一些与 SVG 的重要区别。

- SVG1.1 变换使用用户单位和隐式角度。[3] 尽管 CSS 规范中规定当应用于 SVG 元素时允许使用隐式用户单位，但 CSS 变换中仍然需要使用 CSS 长度并需要指定角度单位。
- SVG1.1 变换是结构型属性，而CSS 变换可以指定在样式表中。样式表声明会覆盖属性值。
- 在 CSS 中，变换类型和左括号之间不能有空格，并且必须使用逗号分隔数字值。
- CSS 变换包含一个单独的属性来指定旋转和缩放的原点。在 SVG 中，旋转原点只是 `rotate()` 函数的一部分，并且我们不能为缩放指定原点。
- CSS 变换还包含 3D 效果。

注 3：指 SVG 变换中不需要指定角度单位，默认单位为“角度”。——译者注

第 7 章

路径

第 4 章中描述过，所有基本形状都是 `<path>` 元素的简写形式。建议使用这些简写形式，因为它们让 SVG 更可读，更加结构化。`<path>` 元素则更通用，它通过指定一系列相互连接的线、圆弧和曲线绘制任意形状的轮廓。这些形状轮廓也像基本形状一样可以填充颜色或者绘制轮廓线。此外，路径（以及简写形式的基本形状）还可以用来定义裁剪区域或者透明遮罩的轮廓，第 10 章会有相关内容介绍。

所有描述轮廓的数据都放在 `<path>` 元素的 `d` 属性中（`d` 是 `data` 的缩写）。路径数据包括单个字符的命令，比如 `M` 表示 `moveto`，`L` 表示 `lineto`，接着是该命令的坐标信息。

7.1 moveto、lineto和closepath

每个路径都必须以 `moveto` 命令开始。

命令字母为大写的 `M`，紧跟着一个使用逗号或空格分隔的 *x* 和 *y* 坐标。这个命令用来设置绘制轮廓的“笔”的当前位置。

`moveto` 命令后面紧跟着一个或多个 `lineto` 命令，用大写 `L` 表示，它的后面也是由逗号或空格分隔的 *x* 和 *y* 坐标。示例 7-1 中有三个路径。第一个绘制了一条线，第二个绘制了一个直角，第三个绘制了两个 30 度的角。当我们使用另一个 `moveto` 命令“重新启用”画笔时，会开始一条新的子路径。我们可以使用逗号或者空格分隔 *x* 和 *y* 坐标，正如示例中三个路径所示。结果如图 7-1 所示。

示例 7-1：使用 moveto 和 lineto

http://oreillymedia.github.io/svg-essentials-examples/ch07/moveto-lineto.html

```
<svg width="150px" height="150px" viewBox="0 0 150 150"
  xmlns="http://www.w3.org/2000/svg">
<g style="stroke: black; fill: none;">
  <!-- 一条线 -->
  <path d="M 10 10 L 100 10"/>

  <!-- 一个直角 -->
  <path d="M 10, 20  L 100, 20  L  100,50"/>

  <!-- 两个30度角 -->
  <path d="M 40 60 L 10, 60 L 40 42.68
    M 60, 60 L 90 60 L 60, 42.68"/>
</g>
</svg>
```

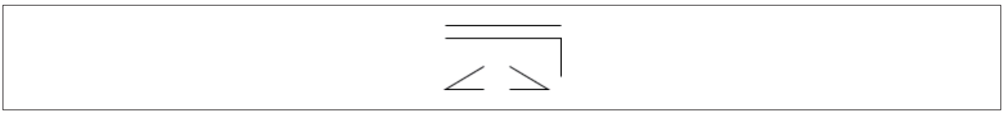


图 7-1：使用 moveto 和 lineto 的结果

仔细观察最后一个路径，它使用了逗号而不是空格来分隔坐标。

值	作　　用
M 40 60	移动画笔到 (40, 60)
L 10 60	绘制一条线到 (10, 60)
L 40 42.68	绘制一条线到 (40, 42.68)
M 60 60	启动一个新的子路径；移动画笔到 (60, 60)——不会绘制线条
L 90 60	绘制一条线到 (90, 60)
L 60 42.68	绘制一条线到 (60, 42.68)

你可能注意到了，路径数据和典型的 XML 属性值看起来不是很像。因为整个路径数据包含在一个属性中，而非每个点或线段都是一个独立的元素，所以当使用 XML 解析器读取 DOM 结构时，路径占用的内存更少。此外，紧凑的路径表示法在传播复杂的图形时不会占用大量的带宽。

如果想用 <path> 绘制矩形，可以采用绘制四条线的方式，也可以先绘制三条线，然后使用用大写 Z 表示的 closepath 命令绘制一条直线回到当前子路径的起点。示例 7-2 中的 SVG 如图 7-2 所示，它展示了绘制所有的边线构成的矩形、使用 closepath 绘制的矩形，以及一个通过开启和闭合两个子路径绘制两个三角形的路径。

示例 7-2：使用 closepath

```
<g style="stroke: black; fill: none;">
  <!-- 四条线形式的矩形 -->
  <path d="M 10, 10 L 40, 10 L 40, 30 L 10, 30 L 10, 10"/>

  <!-- closepath绘制的矩形 -->
  <path d="M 60 10 L 90 10 L 90 30 L 60 30 Z"/>

  <!-- 两个30度角 -->
  <path d="M 40 60 L 10 60 L 40 42.68 Z
     M 60 60 L 90 60 L 60 42.68 Z"/>
</g>
```

仔细观察一下最后一个路径。

值	作　　用
`M 40 60`	移动画笔到 (40, 60)
`L 10 60`	绘制一条线到 (10, 60)
`L 40 42.68`	绘制一条线到 (40, 42.68)
`Z`	通过绘制一条直线到 (40, 60) 来关闭路径并启动子路径
`M 60 60`	启动一个新的子路径；移动画笔到 (60, 60)——不会绘制线条
`L 90 60`	绘制一条线到 (90, 60)
`L 60 42.68`	绘制一条线到 (60, 42.68)
`Z`	通过绘制一条直线到 (60, 60) 来关闭路径并启动子路径

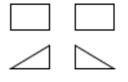


图 7-2：使用 closepath

使用四条线绘制矩形和使用 closepath 命令绘制矩形还有另外一个区别。当关闭路径时，开始线和结束线会被连接到一起，形成一个有样式的连续形状。如果使用粗笔画或者设置 stroke-linecap 以及 stroke-linejoin 效果，区别就很明显了。示例 7-3 使用了一个较大的笔画宽度，图 7-3 展示了结果，并且放大了，以使区别清晰可见。

示例 7-3：独立线条与 closepath

```
<g style="stroke: gray; stroke-width: 8; fill: none;">

   <!-- 四条线形式的矩形 -->
   <path d="M 10 10 L 40 10 L 40 30 L 10 30 L 10 10"/>

   <!-- closepath绘制的矩形 -->
   <path d="M 60 10 L 90 10 L 90 30 L 60 30 Z"/>
</g>
```



图 7-3：独立线条与 closepath 的结果

7.2 相对moveto和lineto

前面的命令都使用大写字母表示，并且坐标都被假定为绝对坐标。如果使用小写命令字母，坐标会被解析为相对于当前的画笔位置。因此，下面两个路径是相同的。

```
<path d="M 10 10 L 20 10 L 20 30 M 40 40 L 55 35"
    style="stroke: black;"/>
<path d="M 10 10 l 10  0 l  0 20 m 20 10 l 15 -5"
    style="stroke: black;"/>
```

如果使用小写 m（moveto）启动路径，它的坐标会被解析为绝对位置，因为它没有参照位置来计算相对位置。本章中所有其他命令都有同样的大小写差别。大写命令的坐标是绝对的，小写命令的坐标是相对的。但是 closepath 命令没有坐标，它的大小写形式效果相同。

7.3 路径的快捷方式

如果说内容是国王，设计是王后，那么带宽效率就是让王宫平稳运行的皇家朝臣。由于任何有意义的绘图都包含由数十个坐标对组成的路径，所以 <path> 元素有一些快捷方式，允许我们使用尽可能少的字节来描绘路径。

7.3.1 水平和垂直lineto命令

水平线和垂直线很常用，足以成为快捷命令。路径可以使用 H 命令加绝对 x 坐标，或者 h 命令加相对 x 坐标，来指定一条水平线。类似地，垂直线可以使用 V 命令加绝对 y 坐标，或者 v 命令加相对 y 坐标来指定。

下表比较了使用简写方式和冗长方式绘制水平和垂直线。

简写形式	等价的冗长形式	效　果
H 20	L 20 current_y	绘制一条到绝对位置 (20, current_y) 的线
h 20	l 20 0	绘制一条到 (current_x + 20,current_y) 的线
V 20	L current_x 20	绘制一条到绝对位置 (current_x,20) 的线
v 20	l 0 20	绘制一条到 (current_x, current_y + 20) 的线

因此，下面的路径绘制了一个宽度为 15 单位、高度为 25 单位的矩形，并且其左上角在坐标 (12, 24) 处。

```
<path d="M 12 24 h 15 v 25 h -15 z"/>
```

7.3.2 路径快捷方式表示法

应用下面两个规则，路径还可以更短。

- 可以在 L 或者 l 后面放多组坐标，正如在 <polyline> 元素中那样。下面六个路径都绘制了如图 7-4 所示的菱形，前三个使用了绝对坐标，后三个使用了相对坐标。其中第三个和第六个中有一个值得关注的点——如果在 moveto 后面放置多对坐标，除了第一对坐标外，剩下的坐标都会被假设为它们跟在一个 lineto 后面。[1]

```
<g style="fill:none; stroke: black">
  <path d="M 30 30 L 55 5 L 80 30 L 55 55 Z"/>
  <path d="M 30 30 L 55 5 80 30 55 55 Z"/>
  <path d="M 30 30 55 5 80 30 55 55 Z"/>
  <path d="m 30 30 l 25 -25 l 25 25 l -25 25 z"/>
  <path d="m 30 30 l 25 -25 25 25 -25 25 z"/>
  <path d="m 30 30 25 -25 25 25 -25 25 z"/>
</g>
```

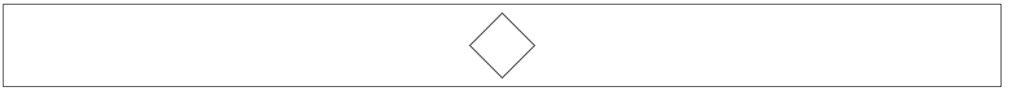


图 7-4：使用路径绘制的菱形

- 所有不必要的空白都可以消除。命令字母后面不需要空白，因为所有的命令都是一个字母。数字和命令之间不需要空白，因为命令字母并不能作为数字的一部分。正数和负数之间也不需要空白，因为负数的前置减号并不能作为正数的一部分。这样，我们就可以进一步缩短前面列出的第三个和第六个路径：

```
<path d="M30 30 55 5 80 30 55 55Z"/>
<path d="m30 30 25-25 25 25-25 25z"/>
```

消除空白规则的另一个例子是，绘制一个宽度为 15 个单位、高度为 25 个单位，左上角在坐标 (12, 24) 的矩形：

```
<path d="M 12 24 h 15 v 25 h -15 z"/><!-- 原始路径 -->
<path d="M12 24h15v25h-15z"/> <!-- 更短的路径 -->
```

7.4 椭圆弧

绘制直线段相对简单，因为路径上的两个点就唯一确定了它们之间的线段。但如果是曲线的话，由于在两个点之间可以绘制无限条曲线，因此我们必须给出额外信息，以在它们之间绘制一条曲线路径。这里要研究的最简单的曲线为椭圆弧，也就是绘制一个连接两个点

注 1：我们还可以在水平 lineto 和垂直 lineto 命令后面放置多个坐标值，但只在使用线标记时才会看到效果，目前我们还没讨论过线标记。H 25 35 45 和 H 45 相同，v 11 13 15 和 v 39 相同。

的椭圆的一部分。

尽管弧是视觉上最简单的曲线，但是指定一条唯一的曲线所需要的信息却是最多的。需要指定的第一部分信息是点所在椭圆的 *x* 半径和 *y* 半径。椭圆的范围可以缩小为两个，正如在图 7-5 中 (a) 部分可以看到的。两个点将两个椭圆划分为 4 个圆弧。其中 (b) 和 (c) 是小于 180 度的弧，(d) 和 (e) 都大于 180 度。看看 (b) 和 (c)，你会发现它们的方向不同，(b) 是按照负角度增加（逆时针）方向绘制的，(c) 是按照正角度增加（顺时针）方向绘制的。同样的关系在 (d) 和 (e) 之间也成立。

别急，这还并没有唯一确定潜在的弧！因为并没有规定说椭圆的 *x* 半径必须平行于 *x* 轴。图 7-5 中的 (f) 部分展示了两个点和它们所在的相对于 *x* 轴旋转了 30 度的椭圆。

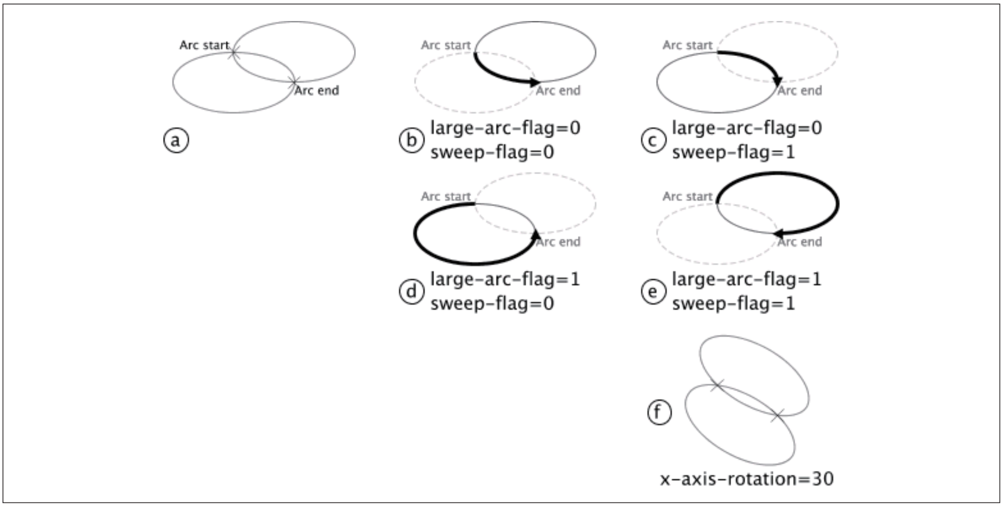


图 7-5：椭圆弧命令

（图 7-5 改编自 W3C SVG 规范的 8.3.8 小节。）

圆弧命令以字母 `A`（绝对坐标的缩写）或者 `a`（相对坐标的缩写）开始，后面紧跟以下 7 个参数。

- 点所在椭圆的 *x* 半径和 *y* 半径。
- 椭圆的 *x* 轴旋转角度 `x-axis-rotation`。
- `large-arc-flag`，如果需要圆弧的角度小于 180 度，其为 0；如果需要圆弧的角度大于或等于 180 度，则为 1。
- `sweep-flag`，如果需要弧以负角度绘制则为 0，以正角度绘制则为 1。
- 终点的 *x* 坐标和 *y* 坐标（起点由最后一个绘制的点或者最后一个 `moveto` 命令确定）。

下面的路径绘制了图 7-5 中 (b) 到 (e) 部分的椭圆弧：

```
<path d="M 125,75 A100,50 0 0,0 225,125"/> <!-- b -->
<path d="M 125,75 A100,50 0 0,1 225,125"/> <!-- c -->
<path d="M 125,75 A100,50 0 1,0 225,125"/> <!-- d -->
<path d="M 125,75 A100,50 0 1,1 225,125"/> <!-- e -->
```

在线示例中，你可以尝试所有圆弧参数来看看它们的作用：

http://oreillymedia.github.io/svg-essentials-examples/ch07/arc.html

这里有一个进一步的示例，让我们来增强一下示例 5-8 中的背景，完成韩国国旗中的阴阳符号。示例 7-4 中用 <ellipse> 元素保持了完整的椭圆，还用路径创建了所需的半圆。结果如图 7-6 所示。

示例 7-4：使用椭圆弧

```
<svg width="400px" height="300px" viewBox="0 0 400 300"
  xmlns="http://www.w3.org/2000/svg">
  <!-- 灰色投影 -->
  <ellipse cx="154" cy="154" rx="150" ry="120" style="fill: #999999;"/>

  <!-- 浅蓝色椭圆 -->
  <ellipse cx="152" cy="152" rx="150" ry="120" style="fill: #cceeff;"/>

  <!-- 浅红色大半圆填充符号的上半部分,其下方“浸入"符号左下方的浅红色小半圆 -->
  <path d="M 302 152 A 150 120, 0, 1, 0, 2 152
      A 75 60, 0, 1, 0, 152 152" style="fill: #ffcccc;"/>

  <!-- 浅蓝色小半圆,填充符号右上方 -->
  <path d="M 152 152 A 75 60, 0, 1, 1, 302 152" style="fill: #cceeff;"/>
</svg>
```

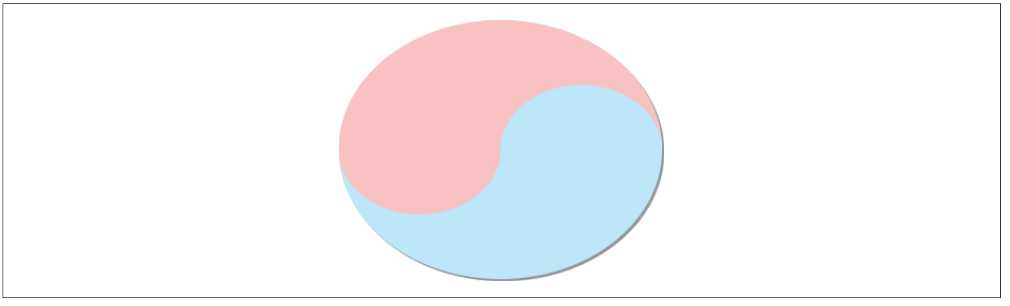


图 7-6：使用椭圆弧的结果



我们不能使用一个路径命令绘制一个完整的椭圆；如果弧形的起点和终点相同，则有无数种方式定位椭圆。SVG 阅读器会跳过这样的圆弧命令。如果你指定的椭圆半径太小，导致不能覆盖起点和终点，则 SVG 阅读器会扩大椭圆直到它足够覆盖起点和终点。

对于如何处理超出范围的参数，请参考弧形实现说明规范（http://www.w3.org/TR/SVG11/implnote.html#ArcImplementationNotes）。

7.5 从其他弧线格式转换

你可能疑惑，为什么不能像其他矢量图形系统那样，通过给椭圆定义一个中心点、x 和 y 半径、起始角度和弧形角度，来指定一个弧形。这是规范中一个简单的方法，适合将弧形绘制为一个单一的对象。矛盾的是，这也正是 SVG 选择一种看似古怪的方法来指定弧形的原因。在 SVG 中，弧形并不能单独存在，它要成为线和曲线连接路径的一部分（比如，一个圆角矩形就是由一系列线和椭圆弧组成的）。因此，通过终点指定弧形是合理的。

然而，有时候我们想要一个独立的半圆（或者更准确地说，半个椭圆）。假设有一个按照如下方式指定的椭圆：

```
<ellipse cx="cx" cy="cy" rx="rx" ry="ry"/>
```

下面是绘制四种可能的半个椭圆的路径（括号中是要计算的代数表达式）：

```
<!-- 北半球 -->
<path d="M (cx - rx) cy
  A rx ry 0 1 1 (cx + rx) cy"/>
<!-- 南半球 -->
<path d="M (cx - rx) cy
  A rx ry 0 1 0 (cx + rx) cy"/>
<!-- 东半球 -->
<path d="M cx (cy - ry)
  A rx ry 0 1 1 cx (cy + ry)"/>
<!-- 西半球 -->
<path d="M cx (cy - ry)
  A rx ry 0 1 0 cx (cy + ry)"/>
```

有时候我们可能想要绘制一个指定了中心和角度的任意弧，并且想要将它转换为 SVG 的终点和范围格式。在另外一些情况下，还可能希望将弧形由 SVG 格式转换为中点和角度格式。第二种情况的数学运算相当复杂，详细信息可查看 SVG 规范。在附录 F 中可以看到这些转换方式的 JavaScript 版本。

7.6 贝塞尔曲线

弧形的特点是整洁，也能达到我们想要的效果，但是很少有人用“优雅”这个词来形容它。如果想要优雅，则需要使用通过二次和三次方程绘图生成曲线。数学家在几百年前就知道这些曲线了，但是绘制它们始终是一个计算很复杂的任务。法国汽车制造商雷诺公司的工程师皮埃尔· 贝塞尔和雪铁龙公司的物理学家和数学家 Paul de Casteljau 改变了这一状况，他们开发并推广了一种计算更简便的方式来生成这些曲线。

如果你用过 Adobe Illustrator 这类绘图程序，可以通过指定两个点以及移动如下图中所示的“控制柄”来绘制这些贝塞尔曲线。控制柄的端点称为控制点，因为它控制着曲线的形状。当我们移动控制柄时，曲线以一种在外行看来非常神秘的方式变化着。Key Point 软件公司

的图形设计师迈克 · 伍德（Mike Woodburn），提出了图 7-7 这种形象地表示控制点和曲线关系的方式：想象一下线是由柔性金属制造的。控制点内部是一个磁铁，与控制点越近，吸引力越强。

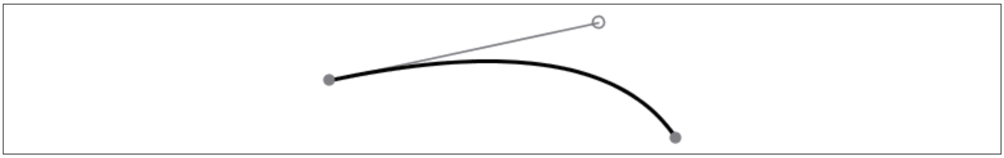


图 7-7：用绘图程序绘制贝塞尔曲线

另一种形象地表示控制点作用的方法以构造曲线的 de Casteljau 方法为基础。下面几节中会使用这种方法。详情请查看介绍得非常清晰的数学部分（http://graphics.cs.ucdavis.edu/~joy/ecs178/Unit-2-Notes/Divide-and-Conquer-Bezier-Curve.pdf）。

7.6.1 二次贝塞尔曲线

最简单的贝塞尔曲线是二次曲线。我们要指定起点、终点和控制点。假设有两个支柱放在线的两个端点，这两个支柱的交点就是控制点。拉紧两个支柱中点的是一根橡皮筋。其中曲线弯曲的地方正好和橡皮筋的中点相连。如图 7-8 所示。

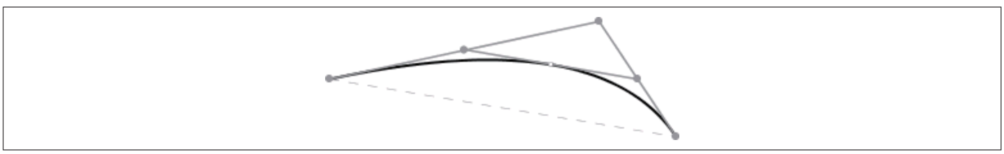


图 7-8：二次贝塞尔曲线

起点 / 终点以及控制点之间的线与曲线的起点 / 终点成切线。曲线首先沿着到达控制点的线，然后弯曲到达中点，再顺着“支柱”线的方向。曲线最终从控制点沿着线向上滑动到终点。Adobe Illustrator 这类程序只会展示一个“支柱”。下次使用这类程序时，可以在心里添加第二个支柱，这样产生的曲线就不那么神秘了。

这些都是理论，现在的实际问题是如何在 SVG 中生成曲线。可以通过在 `<path>` 数据中使用 `Q` 或者 `q` 命令指定一个二次曲线。这个命令后面紧跟着两组指定控制点和终点的坐标。大写命令意味着绝对坐标，小写命令意味着相对坐标。图 7-8 中的曲线起点在 (30, 75)，终点在 (300, 120)，控制点在 (240, 30)，指定在如下所示的 SVG 中。

```
<path d="M30 75 Q240 30, 300 120" style="stroke: black; fill: none;"/>
```

在线示例分别展示了显示和隐藏“支柱”的结果：

http://oreillymedia.github.io/svg-essentials-examples/ch07/quadratic-bezier.html

还可以在二次曲线命令后面指定多组坐标。这会生成一个多边贝塞尔曲线。假设想用 `<path>` 绘制一条从 (30, 100) 到 (100, 100) 并且控制点在 (80, 30) 的曲线，然后绘制一条到 (200, 80) 且控制点在 (130, 65) 的曲线。下面是这个路径的 SVG 示例，其中控制点坐标加粗了。结果如图 7-9 左半部分所示，控制点和线如图右半部分所示。

```
<path d="M30 100 Q 80 30, 100 100, 130 65, 200 80"/>
```

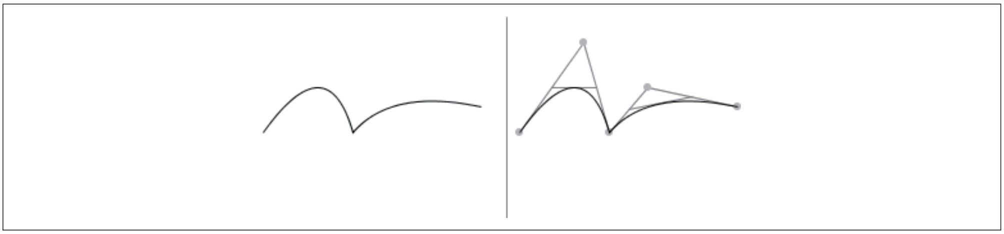


图 7-9：多边二次贝塞尔曲线

你可能心想：“哪里优雅了？该曲线是凹凸不平的。”这个评价是正确的。这是因为曲线相连接并不意味着它们在一起会很好看。这也是为什么 SVG 还提供了流畅的二次曲线命令，用字母 `T` 表示（想要使用相对坐标，就用 `t`）。这个命令后面紧跟的是曲线的下一个端点；如规范所说，控制点会自动计算，方法是“使新的控制点与上一条命令中的控制点相对于当前点中心对称”。

从数学角度来讲，新的控制点 `x2`，`y2` 基于上一条线段的端点 `x`，`y` 和上一个控制点 `x1`，`y1`，按照如下规则计算：

```
x2 = x + (x - x1) = 2 * x - x1
y2 = y + (y - y1) = 2 * y - y1
```

下面是一条从 (30, 100) 到 (100, 100)，控制点在 (80, 30)，然后平滑过渡到 (200, 80) 的曲线。图 7-10 的左半部分展示了这条曲线，右半部分展示了控制点。虚线表示了中心对称的控制点。现在优雅多了！

```
<path d="M30 100 Q 80 30, 100 100 T 200 80"/>
```

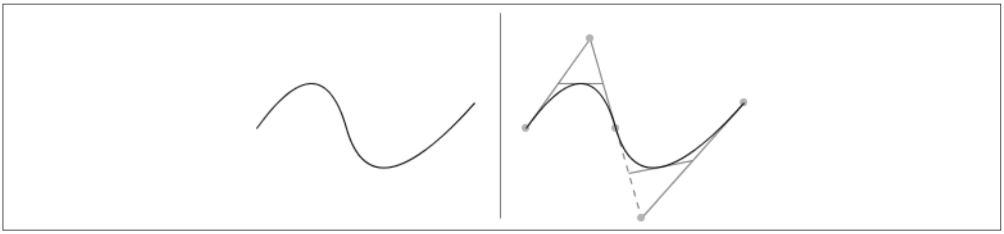


图 7-10：平滑的多边二次贝塞尔曲线

在在线示例中可以体验多边二次贝塞尔曲线：

http://oreillymedia.github.io/svg-essentials-examples/ch07/smooth-quadratic-bezier.html

7.6.2 三次贝塞尔曲线

单个二次贝塞尔曲线有且只有一个顶点，或者每个曲线断都只有一个凹谷。虽然这些曲线比简单的弧线更有用，但是用三次贝塞尔曲线可以做得更好，它可以让同一个曲线图形内既有顶点也有凹谷。换句话说，三次曲线可以包含一个拐点（曲线从该点开始从一个方向往另一个方向弯曲）。

二次曲线和三次曲线之间的区别是三次曲线有两个控制点，每个端点对应一个。生成三次曲线的方式与生成二次曲线的方式类似。从图 7-11 中可以看到，我们绘制了三条线连接端点和控制点 (a)，还连接了它们的中点，生成两条线 (b)。然后再连接 (b) 的中点生成一条线 (c)，它的中点确定了最终曲线上的一个点。注意曲线的起点、终点和中间角度都与控制线成切线（相交）。

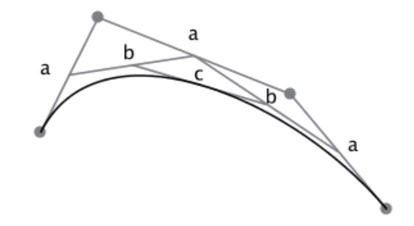


图 7-11：三次贝塞尔曲线

要指定一条三次曲线，使用 C 或者 c 命令。这个命令后面紧跟三组坐标，用来指定起点的控制点、终点的控制点以及端点。和其他所有路径命令一样，大写命令意味着绝对坐标，小写命令意味着相对坐标。上图中的曲线从 (20, 80) 到 (200, 120)，控制点分别在 (50, 20) 和 (150, 60)。SVG 路径如下所示：

```
<path d="M20 80 C 50 20, 150 60, 200 120"
    style="stroke: black; fill: none;"/>
```

根据控制点的关系，还可以绘制很多有趣的曲线（参见图 7-12）。为了让图形整洁，下面只展示了从每个端点到它控制点的线条。

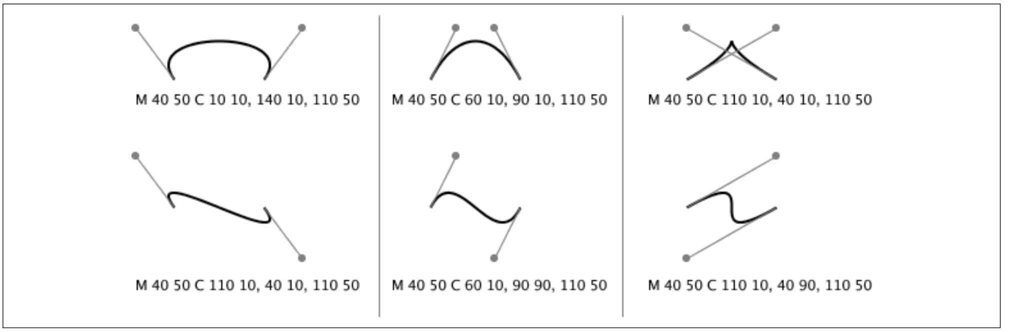


图 7-12：三次曲线控制点组合的结果

可以在在线示例中尝试这些组合或者更多组合：

http://oreillymedia.github.io/svg-essentials-examples/ch07/cubic-bezier.html

和二次曲线一样，也可以通过在三次曲线命令后面指定多组坐标，来构建多条连接在一起的三次曲线。第一条曲线的最后一个点会变成下一条曲线的第一个点，以此类推。这里有一个 <path>，绘制了一条从 (30, 100) 到 (100, 100)，控制点在 (50, 50) 和 (70, 20) 的三次曲线；后面又紧跟着一条曲线，折回到 (65, 100)，控制点在 (110，130) 和 (45, 150) 处。下面是这个路径的 SVG，加粗的是控制点坐标：

```
<path d="M30 100 C 50 50, 70 20, 100 100,
                  110, 130, 45, 150, 65, 100"/>
```

结果如图 7-13 左半部分所示，右半部分展示了控制点和线。

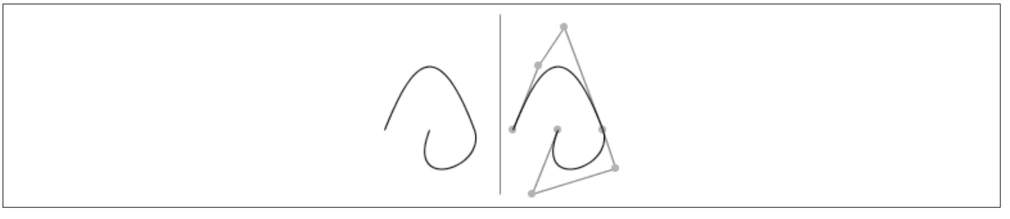


图 7-13：多条连接在一起的三次贝塞尔曲线

如果想要保证曲线之间的连接平滑，可以使用 S 命令（或者如果想要使用相对坐标，就用 s）。在某种程度上，它和二次曲线的 T 命令类似，新的曲线会把上一条曲线的端点作为它的起点，并且它的第一个控制点是上一个终点控制点的中心对称点。我们需要提供的只是曲线的下一个端点的控制点，然后紧跟着的是下一个端点。

这里有一个三次贝塞尔曲线，从 (30,100) 到 (100,100)，控制点为 (50,30) 和 (70,50)。然后它平滑过渡到 (200,80)，使用 (150,40) 作为终点控制点。图 7-14 左半部分展示了曲线，右半部分展示了带有控制点的曲线。虚线展示了中心对称的控制点。

```
<path d="M30 100 C 50 30, 70 50, 100 100 S 150 40, 200 80"/>
```

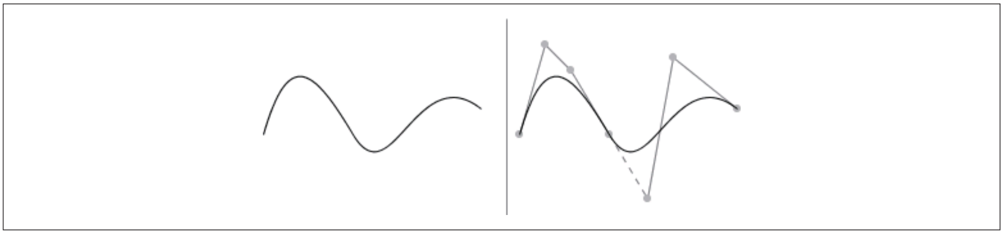


图 7-14：平滑的三次贝塞尔曲线

7.7 路径总结

表 7-1 中，大写命令使用绝对坐标，小写命令使用相对坐标。

表7-1：路径命令

命令	参　数	效　果
`M m`	*x y*	移动到给定坐标
`L l`	*x y*	绘制一条到给定坐标的线。可以提供多组坐标来绘制折线
`H h`	*x*	绘制一条到给定 *x* 坐标的横线
`V v`	*y*	绘制一条到给定 *y* 坐标的竖线
`A a`	*rx ry x-axis-rotation large-arc sweep x y*	绘制一个从当前点到 (*x*,*y*) 的椭圆弧。椭圆上的 *x* 半径为 `rx`，*y* 半径为 `ry`。椭圆旋转 `ry x-axis-rotation` 度。如果圆弧小于 180 度，则 `large-arc` 为 0；如果大于 180 度，则 `large-arc` 为 1。如果圆弧按顺时针方向绘制，则 `sweep` 为 1，否则为 0
`Q q`	*x1 y1 x y*	绘制一条从当前点到 (*x*, *y*)，控制点为 (*x*1, *y*1) 的二次贝塞尔曲线
`T t`	*x y*	绘制一条从当前点到 (*x*, *y*) 的二次贝塞尔曲线。控制点是前一个 Q 命令的控制点的中心对称点。如果没有前一条曲线，当前点会被用作控制点
`C c`	*x1 y1 x2 y2 x y*	绘制一条从当前点到 (*x*, *y*) 的三次贝塞尔曲线，(*x*1, *y*1) 为曲线的开始控制点，(*x*2, *y*2) 为曲线的终点控制点
`S s`	*x2 y2 x y*	绘制一条从当前点到 (*x*, *y*) 的三次贝塞尔曲线，使用 (*x*2, *y*2) 作为新端点的控制点。第一个控制点是前一个 C 命令的终点控制点的中心对称点。如果前一个曲线不存在，当前点会被用作第一个控制点

7.8 路径和填充

4.5 节描述的信息同样适用于路径，路径中不仅可以有相交线，还可以有“缺口”。思考一下示例 7-5 中的路径，两条路径都绘制了嵌套正方形。第一个路径中，按照顺时针方向绘制了两个正方形；第二个路径中，外部正方形按顺时针方向绘制，内部正方形按逆时针方向绘制。

示例 7-5：在路径中使用不同的 fill-rule 值

```
<!-- 顺时针方向的路径 -->
<path d="M 0 0, 60 0, 60 60, 0 60 Z
    M 15 15, 45 15, 45 45, 15 45Z"/>

<!-- 外部路径为顺时针方向,内部路径为逆时针方向 -->
<path d="M 0 0, 60 0, 60 60, 0 60 Z
    M 15 15, 15 45, 45 45, 45 15Z"/>
```

图 7-15 展示了使用 `fill-rule` 为 `nonzero` 时的不同，确定一个点是在路径的内部还是外部时参考线条的方向。而使用 `fill-rule=evenodd` 为两个路径生成了相同的结果，它参考的是交叉线的总数，但是忽略它们的方向。

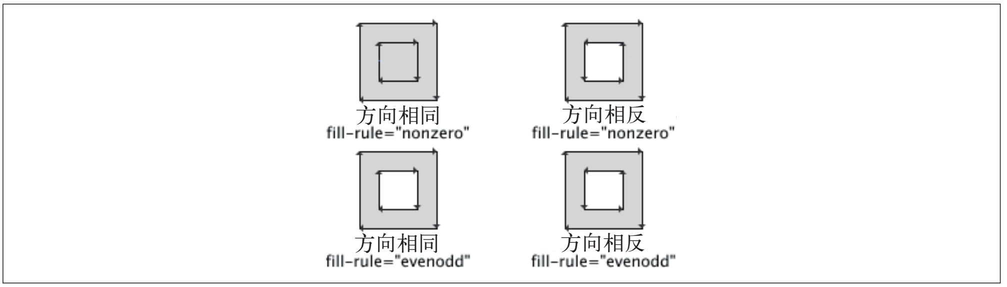


图 7-15：使用不同 fill-rule 的结果

7.9 <marker>元素

思考下面的路径，其中用了两条线和一个椭圆弧绘制如图 7-16 所示的圆角：

```
<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="fill: none; stroke: black;"/>
```

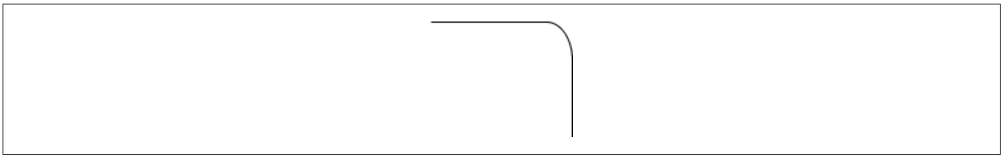


图 7-16：线和圆弧

假设想要通过在开始位置放一个圆，在结束位置放一个实心三角形，以及在其他顶点放一些箭头来标记路径的方向，如图 7-17 所示的那样。要实现这一效果，需要构建三个 `<marker>` 元素，然后让 `<path>` 元素引用它们。

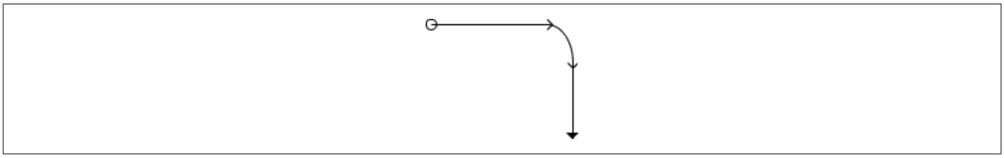


图 7-17：带有标记的线和圆弧

在示例 7-6 中，第一步添加了圆形标记。一个标记就是一个“独立的”图形，它拥有自己私有的坐标，因此必须在开始标记 <marker> 中指定它的 markerWidth 和 markerHeight。后面是绘制标记需要的 SVG 元素，最后以结束标记 </marker> 结束。<marker> 元素自身不会显示，但是可以把它放到 <defs> 元素中，因为它是存放可复用元素的。

因为我们想要圆形位于路径的开始位置，因此给 <path> 的 style 属性添加了一个 marker-start。[2] 这个属性的值是刚才创建的 <marker> 元素的 URL。

示例 7-6：圆形标记初试

```
<defs>
<marker id="mCircle" markerWidth="10" markerHeight="10">
    <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
</marker>
</defs>

<path  d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker-start: url(#mCircle);
    fill: none; stroke: black;"/>
```

结果如图 7-18 所示，但结果并不完全符合预期。

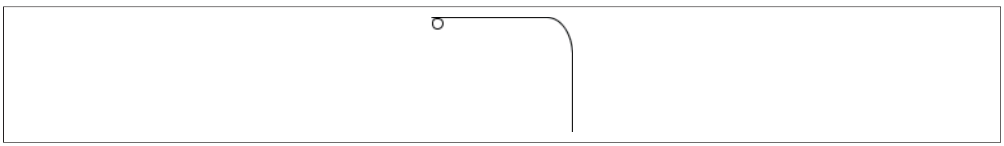


图 7-18：放置错误的圆形标记

圆形标记显示在错误位置的原因是，默认情况下，开始标记的 (0,0) 点与路径的开始坐标对齐。示例 7-7 中添加了 refX 和 refY 属性指定哪个坐标（标记的坐标系统中）与路径的开始坐标对齐。一旦添加好之后，圆形标记就会精确地显示在图 7-19 中期望的位置。

示例 7-7：正确放置圆形标记

```
<marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
    <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
</marker>
```

注 2：是的，标记被认为是表现而不是结构的一部分。这是灰色地带之一，你可以支持任何一种观点。

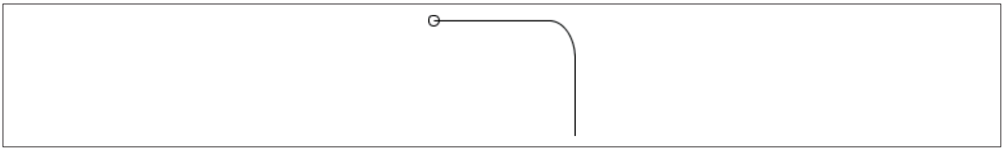


图 7-19：正确放置的圆形标记

基于以上信息，现在可以编写示例 7-8 了，其中添加了三角形标记并在路径的 marker-end 中引用它。然后可以添加箭头标记并在 marker-mid 中引用它。marker-mid 会附加给除路径起点和终点以外的每个顶点。注意，还设置了 refX 和 refY 属性，因此箭头较宽的一端能与中间的顶点对齐，而实心三角形的尖角与结束顶点对齐。图 7-20 展示了结果，但是只正确绘制了第一个标记，其他标记并不正确。

示例 7-8：尝试使用三个标记

```
<defs>
    <marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
        <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mArrow" markerWidth="4" and markerHeight="8"
        refX="0" refY="4">
        <path d="M 0 0 4 4 0 8" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mTriangle" markerWidth="5" markerHeight="10"
        refX="5" refY="5">
        <path d="M 0 0 5 5 0 10 Z" style="fill: black;"/>
    </marker>
</defs>

<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker-start: url(#mCircle);
        marker-mid: url(#mArrow);
        marker-end: url(#mTriangle);
        fill: none; stroke: black;"/>
```

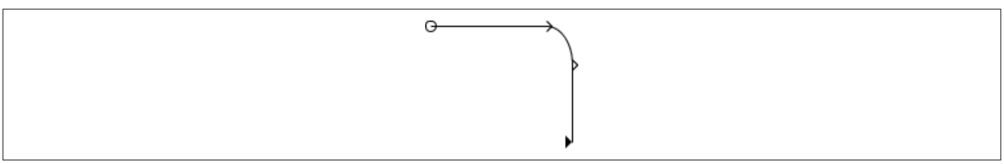


图 7-20：错误定向的标记

为了获取想要的效果，必须明确设置标记的 orient 属性为 auto。这会让标记自动旋转来匹配路径的方向。[3]（也可以指定度数，此时标记始终按照指定的度数旋转。）示例 7-9 中的

注 3：确切地讲，旋转角度是线条进入标记图形和线条退出标记图形时的角度平均值。

标记会生成图 7-17 所示的结果。无需确定圆的方向；不管怎么旋转，它看起来都一样。

示例 7-9：正确定向的标记

```
<defs>
    <marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
        <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mArrow" markerWidth="6" markerHeight="10"
        refX="0" refY="4" orient="auto">
        <path d="M 0 0 4 4 0 8" style="fill: none; stroke: black;"/>
    </marker>

    <marker id="mTriangle" markerWidth="5" markerHeight="10"
        refX="5" refY="5" orient="auto">
        <path d="M 0 0 5 5 0 10 Z" style="fill: black;"/>
    </marker>
</defs>

<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker-start: url(#mCircle);
        marker-mid: url(#mArrow);
        marker-end: url(#mTriangle);
        fill: none; stroke: black;"/>
```

另一个有用的属性是 markerUnits。如果设置为 strokeWidth，那么标记的坐标系统会被设定为单位等于笔画宽度。标记会与笔画宽度成正比，这是它的默认行为，通常也是我们想要的。如果设置这个属性为 userSpaceOnUse，标记的坐标系统会被假定为和引用该标记的对象的坐标系统一样。不管笔画宽度为多少，标记都会保持相同的尺寸。

7.10 标记记录

如果想要路径的起点、中间和终点都使用相同的标记，无需指定所有的 marker-start、marker-mid 和 marker-end 属性。只需使用 marker 属性引用想要的标记即可。因此，如果想要所有的顶点都有一个圆形标记，如图 7-21 所示，编写示例 7-10 所示的 SVG 即可。

示例 7-10：为所有顶点使用一个标记

```
<defs>
    <marker id="mCircle" markerWidth="10" markerHeight="10"
        refX="5" refY="5">
        <circle cx="5" cy="5" r="4" style="fill: none; stroke: black;"/>
    </marker>
</defs>

<path d="M 10 20 100 20 A 20 30 0 0 1 120 50 L 120 110"
    style="marker: url(#mCircle); fill: none; stroke: black;"/>
```

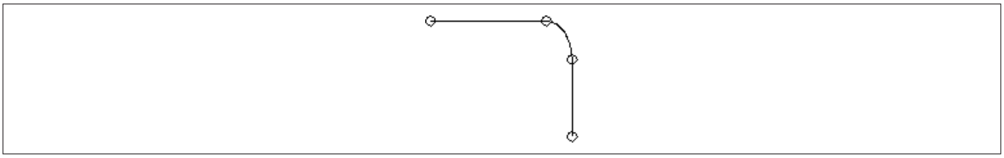


图 7-21：为所有顶点使用一个标记

还可以在 <marker> 元素上设置 viewBox 和 preserveAspectRatio 属性，以更好地控制它的显示效果。例如，可以使用 viewBox 定义网格，让 (0,0) 坐标位于标记的中心；我们可能想要使用这种方式，而不是使用 refX 和 refY。viewBox 和 preserveAspectRaio 属性的工作方式已在 3.3 节和 3.4 节介绍过。

可以在 <polygon>、<polyline> 或者 <line> 元素以及 <path> 中引用 <marker>。

你可能有过这种想法："如果标记中可以有路径，那么该路径中是否也可以有标记？"答案是"可以"，但是第二个标记必须适配由第一个标记的 markerWidth 和 markerHeight 建立的矩形。记住，我们只是说这样做是可行的，但并不鼓励这样做。如果需要这种效果，最好将从属标记也作为主标记的一部分，而不是尝试嵌套标记。

确保没有为标记定义自己作为从属标记。用如下的 CSS 规则为所有的路径添加一个星形标记时，可能会发生这种情况。

```
path { marker: url(#star) }
```

如果 id 为 star 的 <marker> 元素中也有一个 <path>，这个路径会无限循环地引用标记自身。为了防止这种情况发生，要添加一条 CSS 规则，说明不要给星形标记中的路径添加任何标记：

```
path {marker: url(#star)}
marker#star path {marker: none}
```

第 8 章

图案和渐变

到目前为止，我们只用了纯色来为图形对象填充颜色和绘制轮廓。除了纯色以外，我们还可以使用图案和渐变来填充图形或者绘制图形轮廓。这也是本章要讨论的内容。

8.1　图案

要使用图案，首先要定义一个水平或者垂直方向重复的图形对象，然后用它填充另一个对象或者作为笔画使用。这个图形对象被称作 tile（瓷砖），因为使用图案填充对象的行为很像在地面上铺瓷砖的过程。本节，我们会把示例 8-1 中使用 SVG 绘制的二次曲线作为图案。灰色的轮廓清晰地显示了它的面积（20 乘 20 用户单位）。

示例 8-1：图案的路径

```
<path d="M 0 0 Q 5 20 10 10 T 20 20"
    style="stroke: black; fill: none;"/>
<path d="M 0 0 h20 v20 h-20 z"
    style="stroke: gray; fill: none;"/>
```

图 8-1 是放大后的图案，方便查看细节。

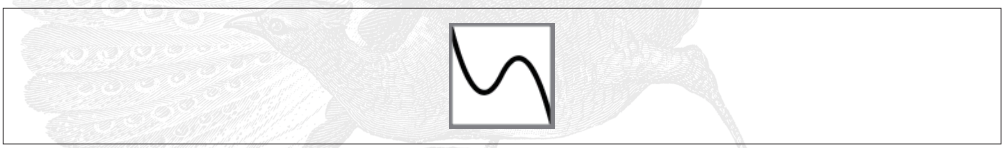


图 8-1：放大后的图案

8.1.1 patternUnits

要创建一个图案，必须使用 `<pattern>` 元素包裹描述图案的 `<path>` 元素，然后确定一些事情。第一个需要确定的是希望如何排列图案，这一点体现在 `patternUnits` 属性上。是想让每个图案填充对象的一定百分比，还是想以相同大小的图案平铺，而无论填充对象的尺寸多大？

如果希望图案的尺寸基于对象的大小计算，需要指定图案左上角的 `x` 和 `y` 坐标，以及其 `width` 和 `height`（百分比或者 0 到 1 之间的小数），然后设置 `patternUnits` 属性为 `objectBoundingBox`（边界框）。对象的边界框是一个完全包裹图形对象的最小矩形。示例 8-2 创建了一个简单的图案，它会在它所填充的对象中水平和垂直方向各重复 5 次。

示例 8-2：设置 patternUnits 为 objectBoundingBox

```
<defs>
<pattern id="tile" x="0" y="0" width="20%" height="20%"
    patternUnits="objectBoundingBox">
<path d="M 0 0 Q 5 20 10 10 T 20 20"
    style="stroke: black; fill: none;"/>
<path d="M 0 0 h 20 v 20 h -20 z"
    style="stroke: gray; fill: none;"/>
</pattern>
</defs>

<rect x="20" y="20" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
<rect x="135" y="20" width="70" height="80"
    style="fill: url(#tile); stroke: black;"/>
<rect x="220" y="20" width="150" height="130"
    style="fill: url(#tile); stroke: black;"/>
```

图 8-2 中，左侧矩形的宽高为 100 用户单位，正好容纳 5 个宽高为 20 用户单位的图案。中间矩形的宽高不够完全展示任意一个图案，因此它们被截断了。右侧的矩形中，产生了额外的间隙，因为其宽高超过一个图案所需空间的 5 倍。由于图案设置了 `x` 和 `y` 值为 0，因此在上面的例子中左上角都恰好为矩形的左上角。

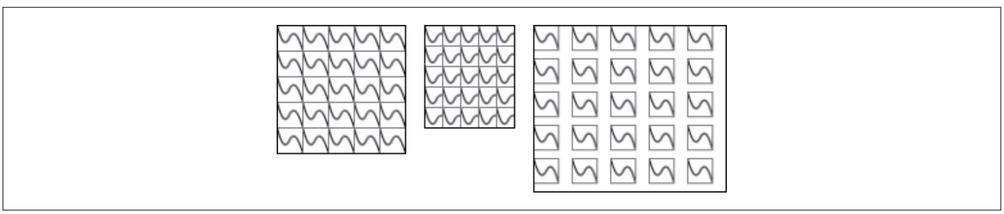


图 8-2：基于 objectBoundingBox 排列的图案

你可能会惊讶地发现，上面的平铺方式和大多数图形设计程序都不一样。标准的图形编辑程序会在画布上将图案一个接一个地放置，而无管尺寸是多少。这些图案之间不会有额外的间隙，并且只在所填充对象的边缘发生进行裁剪。如果这种行为是你想要的，则必须设

置 patternUnits 属性值为 userSpaceOnUse，还要指定 x 和 y 坐标，以及按用户单位指定图案的 width 和 height。示例 8-3 使用了同样的图案，还精确地设置它的宽度和高度为 20 用户单位。

示例 8-3：改变 patternUnits 为 userSpaceOnUse

http://oreillymedia.github.io/svg-essentials-examples/ch08/patternunits.html

```
<defs>
<pattern id="tile" x="0" y="0" width="20" height="20"
    patternUnits="userSpaceOnUse">
<path d="M 0 0 Q 5 20 10 10 T 20 20"
    style="stroke: black; fill: none;"/>
<path d="M 0 0 h 20 v 20 h -20 z"
    style="stroke: gray; fill: none;"/>
</pattern>
</defs>

<rect x="20" y="20" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
<rect x="135" y="20" width="70" height="80"
    style="fill: url(#tile); stroke: black;"/>
<rect x="220" y="20" width="150" height="130"
    style="fill: url(#tile); stroke: black;"/>
```

图 8-3 中，三个矩形中的图案尺寸都是固定的。但是它们的对齐方式取决于所在的坐标系统。例如，中间矩形的 *x* 坐标并不是 20 的倍数，因此矩形的左上角不能与图案的左上角重合（但是顶部是对齐的，因为三个矩形的 *y* 坐标都是特意选择的，都是 20 的倍数）。

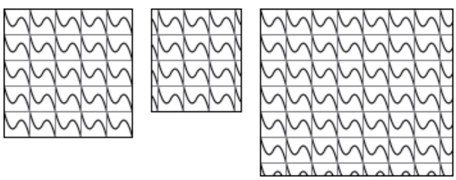


图 8-3：使用 userSpaceOnUse 设置图案间隔



如果没有指定 patternUnits 值，默认为 objectBoundingBox。

8.1.2 patternContentUnits

接下来必须要确定的是用什么单位表达图案数据本身。默认情况下，patternContentUnits 属性值为 userSpaceOnUse。如果设置属性值为 objectBoundingBox，则路径本身的数据点会基于被填充的对象来确定。示例 8-4 中的 SVG 结果如图 8-4。

如果 patternContentUnits 使用 userSpaceOnUse，那么图案的边界框左上角应该在原点 (0, 0) 位置。

如果使用 objectBoundingBox，则需要在图案中减小 stroke-width 的值。图案的宽度也会以被填充对象的边界框作为参考，而不会使用用户单位，因此 stroke-width 为 1 会覆盖整个图案。在这个例子中，笔画宽度被设置为 0.01，也就是被填充对象边界框宽度和高度均值的 1%。

示例 8-4：设置 patternContentUnits 为 objectBoundingBox

```
<defs>
<pattern id="tile"
    patternUnits="objectBoundingBox"
    patternContentUnits="objectBoundingBox"
     x="0" y="0" width=".2" height=".2">
    <path d="M 0 0 Q .05 .20 .10 .10 T .20 .20"
        style="stroke: black; fill: none; stroke-width: 0.01;"/>
    <path d="M 0 0 h 0.2 v 0.2 h-0.2z"
        style="stroke: black; fill: none; stroke-width: 0.01;"/>
</pattern>

</defs>

<g transform="translate(20, 20)">
<rect x="0" y="0" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
</g>

<g transform="translate(135, 20)">
<rect x="0" y="0" width="70" height="80"
    style="fill: url(#tile); stroke: black;"/>
</g>

<g transform="translate(220, 20)">
<rect x="0" y="0" width="150" height="130"
    style="fill: url(#tile); stroke: black;"/>
</g>
```

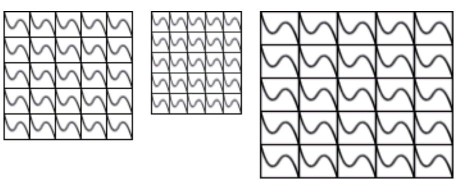


图 8-4：设置 patternContentUnits 为 objectBoundingBox

如果想缩小现有的图形对象当作图案，则使用 viewBox 属性来缩放更容易。指定 viewBox

会覆盖任何 patternContentUnits 信息。另一种做法是像 3.4 节中所描述的那样，设置 preserveAspectRatio 属性。示例 8-5 使用了图 7-13 中的三次贝塞尔曲线的缩小版作为图案。stroke-width 被设置为 5；否则当缩小 SVG 时，在图 8-5 中将看不到图案。

示例 8-5：使用 viewBox 缩放图案

```
<defs>
<pattern id="tile"
    patternUnits="userSpaceOnUse"
    x="0" y="0" width="20" height="20"
    viewBox="0 0 150 150">
    <path d="M30 100 C 50 50, 70 20, 100 100,
                  110, 130, 45, 150, 65, 100"
        style="stroke: black; stroke-width: 5; fill: none;"/>
</pattern>
</defs>

<rect x="20" y="20" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
```

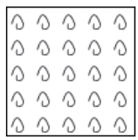


图 8-5：使用 viewBox 缩放后的图案

8.1.3 图案嵌套

同样，你可能会想到："如果对象可以使用图案填充，那么图案能够使用图案填充吗？"答案是肯定的。和标记嵌套完全不同（很少需要嵌套标记），如果不使用图案嵌套，有些效果很难实现。示例 8-6 创建了一个使用圆填充的矩形，这些圆都使用横向条纹填充。这产生了一种不常见，但是有效的、带虚线的条纹效果，如图 8-6 所示。

示例 8-6：图案嵌套

```
<defs>
  <pattern id="stripe"
    patternUnits="userSpaceOnUse"
    x="0" y="0" width="6" height="6">
    <path d="M 0 0 6 0"
      style="stroke: black; fill: none;"/>
  </pattern>

  <pattern id="polkadot"
    patternUnits="userSpaceOnUse"
    x="0" y="0" width="36" height="36">
    <circle cx="12" cy="12" r="12"
      style="fill: url(#stripe); stroke: black;"/>
```

```
    </pattern>
  </defs>

  <rect x="36" y="36" width="100" height="100"
      style="fill: url(#polkadot); stroke: black;"/>
```

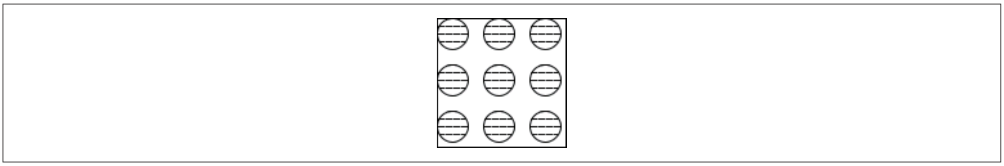


图 8-6：图案中的图案

8.2 渐变

我们可以使用渐变填充对象，也就是从一个颜色平滑地过渡到另一个，而不是使用纯色填充对象。渐变可以是线性的，即颜色沿着直线过渡；也可以是径向的，即颜色从中心点向外辐射（发散）过渡。

8.2.1 linearGradient元素

线性渐变就是一系列颜色沿着一条直线过渡。在特定的位置指定想要的颜色，被称作渐变点（gradient stop）。渐变点是渐变结构的一部分，颜色是表现的一部分。示例 8-7 中的 SVG 展示了一个由金黄色平滑过渡到青色的渐变填充的矩形。结果如图 8-7 所示。

示例 8-7：简单的双色渐变

http://oreillymedia.github.io/svg-essentials-examples/ch08/linear_gradient.html

```
<defs>
  <linearGradient id="two_hues">
    <stop offset="0%" style="stop-color: #ffcc00;"/>
    <stop offset="100%" style="stop-color: #0099cc;"/>
  </linearGradient>
</defs>

<rect x="20" y="20" width="200" height="100"
  style="fill: url(#two_hues);  stroke: black;"/>
```

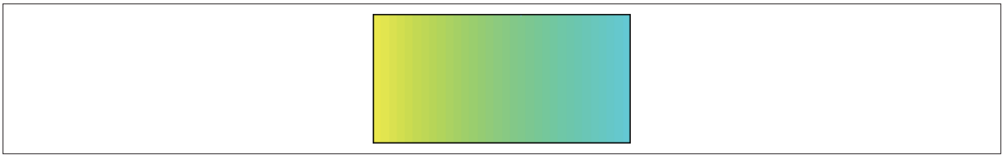


图 8-7：简单的双色渐变

1. <stop>元素

让我们仔细看看 <stop> 元素。它有两个必要属性：offset 和 stop-color。offset 用于确定线上哪个点的颜色应该等于 stop-color。offset 的值使用 0 到 100% 之间的百分比或者 0 到 1.0 之间的小数表示。虽然在 0% 和 100% 位置设置渐变点并不是必须的，但是通常我们都会这么做。这里的 stop-color 被指定在 style 中，但是也可以指定它为独立属性。示例 8-8 展示了一个稍微复杂一点的线性渐变，0% 位置的颜色为金黄色，33.3% 位置的颜色为紫红色，100% 位置的颜色为淡绿色。结果如图 8-8 所示。

示例 8-8：三色渐变

http://oreillymedia.github.io/svg-essentials-examples/ch08/three_stop_gradient.html

```
<defs>
  <linearGradient id="three_stops">
    <stop offset="0%" style="stop-color: #ffcc00;"/>
    <stop offset="33.3%" style="stop-color: #cc6699"/>
    <stop offset="100%" style="stop-color: #66cc99;"/>
  </linearGradient>
</defs>

<rect x="20" y="20" width="200" height="100"
    style="fill: url(#three_stops); stroke: black;"/>
```

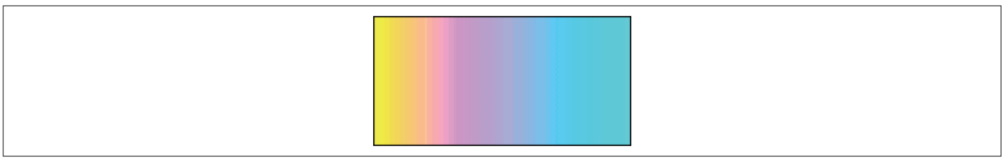


图 8-8：三色渐变

指定渐变点颜色的时候还可以使用 stop-opacity 属性，值为 1 表示完全不透明，为 0 则表示完全透明。示例 8-9 创建了一个快速淡出到中点位置，然后向终点褪色的渐变。其结果如图 8-9 所示。

示例 8-9：带有三个 opacity 的渐变

http://oreillymedia.github.io/svg-essentials-examples/ch08/stop_opacity.html

```
<defs>
  <linearGradient id="three_opacity_stops">
    <stop offset="0%" style="stop-color: #906; stop-opacity: 1.0"/>
    <stop offset="50%" style="stop-color: #906; stop-opacity: 0.3"/>
    <stop offset="100%" style="stop-color: #906; stop-opacity: 0.10"/>
  </linearGradient>
</defs>

<rect x="20" y="20" width="200" height="100"
    style="fill: url(#three_opacity_stops); stroke: black;"/>
```

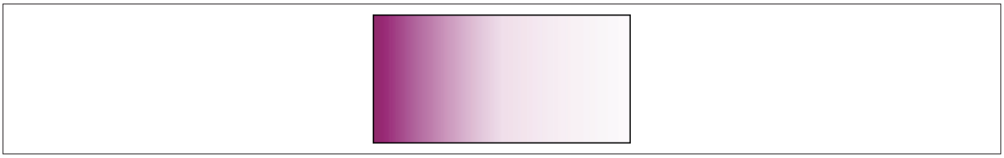


图 8-9：使用 stop-opacity 的渐变

2. 定义线性渐变的方向

线性渐变的默认行为是沿着水平线从对象的左侧向右侧过渡。如果想要颜色沿着竖线或者有角度的线条过渡，就必须使用 x1 和 y1 以及 x2 和 y2 属性指定渐变的起点和终点。默认情况下，它们也使用 0% 到 100% 的百分比或者 0 到 1 的小数表示。示例 8-10 用相同的渐变点建立了一个水平渐变、垂直渐变和对角（线）渐变。这个例子使用 xlink:href 属性引用原始的从左到右的渐变，而不是将渐变点复制给每个 <linearGradient> 元素。这样渐变点会被继承，但是 *x* 坐标和 *y* 坐标会被每个独立的渐变重写。示例 8-10 中的 SVG 结果如图 8-10 所示，SVG 并不包括图中的箭头。

示例 8-10：定义线性渐变的方向

http://oreillymedia.github.io/svg-essentials-examples/ch08/transition_line.html

```
<defs>
<linearGradient id="three_stops">
  <stop offset="0%" style="stop-color: #ffcc00;"/>
  <stop offset="33.3%" style="stop-color: #cc6699;"/>
  <stop offset="100%" style="stop-color: #66cc99;"/>
</linearGradient>

<linearGradient id="right_to_left"
  xlink:href="#three_stops"
  x1="100%" y1="0%" x2="0%" y2="0%"/>

<linearGradient id="down"
  xlink:href="#three_stops"
  x1="0%" y1="0%" x2="0%" y2="100%"/>

<linearGradient id="up"
  xlink:href="#three_stops"
  x1="0%" y1="100%" x2="0%" y2="0%"/>

<linearGradient id="diagonal"
  xlink:href="#three_stops"
  x1="0%" y1="0%" x2="100%" y2="100%"/>
</defs>

<rect x="40" y="20" width="200" height="40"
  style="fill: url(#three_stops); stroke: black;"/>

<rect x="40" y="70" width="200" height="40"
  style="fill: url(#right_to_left); stroke: black;"/>
```

```
<rect x="250" y="20" width="40" height="200"
  style="fill: url(#down); stroke: black;"/>

<rect x="300" y="20" width="40" height="200"
  style="fill: url(#up); stroke: black;"/>

<rect x="40" y="120" width="200" height="100"
  style="fill: url(#diagonal); stroke: black;"/>
```

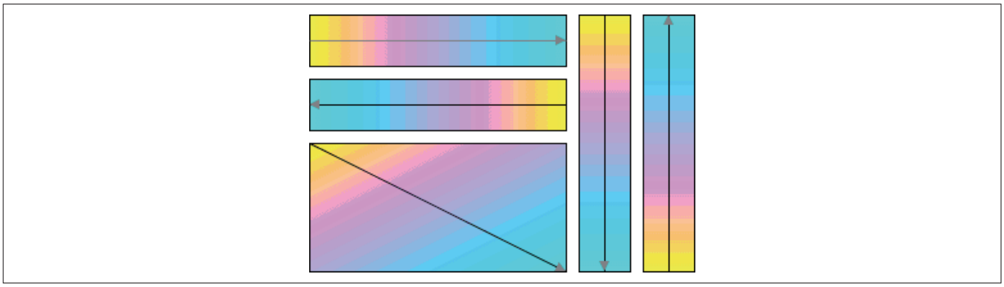


图 8-10：定义线性渐变的方向

> 如果想要使用用户坐标空间而不是百分比指定渐变方向，设置 `gradientUnits` 为 `userSpaceOnUse` 而不是默认值 `objectBoundingBox` 即可。

3. `spreadMethod`属性

指定过渡方向时并不一定要从对象的一角到另一角。如果我们指定从 (20%, 30%) 到 (40%, 80%) 会发生什么？对象中在指定范围之外的部分会发生什么？可以设置 `spreadMethod` 属性为下列值之一。

- `pad`

  起始和结束渐变点会扩展到对象的边缘。

- `repeat`

  渐变会重复起点到终点的过程，直到填充满整个对象。

- `reflect`

  渐变会按终点到起点、起点到终点的排列重复，直到填充满整个对象。

图 8-11 中左侧的正方形展示了 pad 效果，中间的正方形展示了 repeat 效果，右侧的正方形展示了 reflect 效果。示例 8-11 的图中，正方形中原始的渐变被加上了一条线，这样效果会更明显。

示例 8-11：`spreadMethod` 值的线性渐变效果

http://oreillymedia.github.io/svg-essentials-examples/ch08/spread_method.html

```
<defs>
<linearGradient id="partial"
  x1="20%" y1="30%" x2="40%" y2="80%">
  <stop offset="0%" style="stop-color: #ffcc00;"/>
  <stop offset="33.3%" style="stop-color: #cc6699;"/>
  <stop offset="100%" style="stop-color: #66cc99;"/>
</linearGradient>

<linearGradient id="padded"
  xlink:href="#partial"
  spreadMethod="pad"/>

<linearGradient id="repeated"
  xlink:href="#partial"
  spreadMethod="repeat"/>

<linearGradient id="reflected"
  xlink:href="#partial"
  spreadMethod="reflect"/>

<line id="show-line" x1="20" y1="30" x2="40" y2="80"
  style="stroke: white;"/>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#padded); stroke: black;"/>
<use xlink:href="#show-line" transform="translate(20, 20)"/>

<rect x="130" y="20" width="100" height="100"
  style="fill: url(#repeated); stroke: black;"/>
<use xlink:href="#show-line" transform="translate(130, 20)"/>

<rect x="240" y="20" width="100" height="100"
  style="fill: url(#reflected); stroke: black;"/>
<use xlink:href="#show-line" transform="translate(240, 20)"/>
```

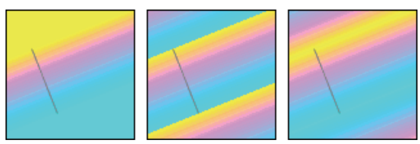


图 8-11：线性渐变的 spreadMethod 值：pad、repeat 和 reflect

8.2.2 radialGradient元素

另一种渐变类型是径向渐变，每个渐变点表示一个圆形路径，从中心点向外扩散。[1] 它的设置方式和线性渐变大致相同。示例 8-12 设置了一个三色渐变：橙色、绿色和紫色。结果如图 8-12 所示。

注 1：如果填充对象的边界框不是正方形的，过渡路径会变成椭圆形来匹配边界框的长宽比。

示例 8-12：三色径向渐变

http://oreillymedia.github.io/svg-essentials-examples/ch08/three_stop_radial.html

```
<defs>
  <radialGradient id="three_stops">
    <stop offset="0%" style="stop-color: #f96;"/>
    <stop offset="50%" style="stop-color: #9c9;"/>
    <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#three_stops); stroke: black;"/>
```

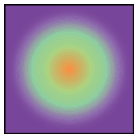


图 8-12：三色径向渐变

1. 定义径向渐变的范围

径向渐变的范围不是使用线条确定哪里是点 0% 和 100%，而是由圆形（圈）确定，其中中心点为 0%，外圆周定义了点 100%。我们使用 cx（中心点 *x* 坐标）、cy（中心点 *y* 坐标）以及 r（半径）属性定义外圆。所有这些属性值都是对象外边框的百分比，默认值都为 50%。示例 8-13 绘制了一个径向渐变的正方形，中心点在正方形的左上角，外边缘在右下方。结果如图 8-13 所示。

示例 8-13：为径向渐变设置范围

http://oreillymedia.github.io/svg-essentials-examples/ch08/radial_limits.html

```
<defs>
  <radialGradient id="center_origin"
    cx="0%" cy="0%" r="141%">
      <stop offset="0%" style="stop-color: #f96;"/>
      <stop offset="50%" style="stop-color: #9c9;"/>
      <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#center_origin); stroke: black;"/>
```



图 8-13：定义径向渐变的范围

在前面的例子中，radialGradient 的 r 被设置为 141%，而不是 100%。这是因为用来测量半径的单位是对象边界框宽度和高度的平均值，而不是框的对角线。正方形对角线与边长的比例是 2 的平方根，即 1.41。

0% 点也被称作焦点，默认为 100% 处渐变点所在圆的圆心。如果希望点 0% 在其他地方而不是圆心，必须改变 fx 和 fy 属性。圆的焦点应该建立在 100% 处渐变点所在圆的内部。如果不是，SVG 阅读器会自动把焦点移到该圆的焦点。

示例 8-14 中，圆的中心点在原点位置，半径为 100%，但是焦点在 (50%,50%) 位置。正如在图 8-14 中可以看到的，这产生一种移动“中心点”的视觉效果。

示例 8-14：设置径向渐变的焦点

http://oreillymedia.github.io/svg-essentials-examples/ch08/radial_focus.html

```
<defs>
  <radialGradient id="focal_set"
    cx="0%" cy="0%" fx="50%" fy="50%" r="100%">
      <stop offset="0%" style="stop-color: #f96;"/>
      <stop offset="50%" style="stop-color: #9c9;"/>
      <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#focal_set); stroke: black;"/>
```



图 8-14：设置径向渐变的焦点

<radialGradient> 的范围设置属性的默认值如下所示。

属　性	默认值
cx	50%（对象边界框的水平中心点）
cy	50%（对象边界框的垂直中心点）
r	50%（对象边界框宽度 / 高度的一半）
fx	和 cx 一样
fy	和 cy 一样

如果想用用户空间坐标而不是百分比确定圆的范围，要设置 gradientUnits 为 userSpaceOnUse，而不是默认值 objectBoundingBox。

2. 径向渐变的spreadMethod属性

如果绘制的范围没有到达对象的边缘，可以设置 spreadMethod 属性为前面“spreadMethod 属性”中所描述的 pad、repeat 或者 reflect 三个值之一，来按照期望的方式填补剩下的空白。示例 8-15 包含了这三种效果，图 8-15 中左侧的正方形展示了 pad 渐变效果，中间的正方形为 repeat 渐变效果，右侧的正方形则呈现了 reflect 渐变效果。

示例 8-15：径向渐变 spreadMethod 的效果演示

http://oreillymedia.github.io/svg-essentials-examples/ch08/radial_spread_method.html

```
<defs>
  <radialGradient id="three_stops"
    cx="0%" cy="0%" r="70%">
      <stop offset="0%" style="stop-color: #f96;"/>
      <stop offset="50%" style="stop-color: #9c9;"/>
      <stop offset="100%" style="stop-color: #906;"/>
  </radialGradient>

  <radialGradient id="padded" xlink:href="#three_stops"
    spreadMethod="pad"/>
  <radialGradient id="repeated" xlink:href="#three_stops"
    spreadMethod="repeat"/>
  <radialGradient id="reflected" xlink:href="#three_stops"
    spreadMethod="reflect"/>
</defs>

<rect x="20" y="20" width="100" height="100"
  style="fill: url(#padded); stroke: black;"/>
<rect x="130" y="20" width="100" height="100"
  style="fill: url(#repeated); stroke: black;"/>
<rect x="240" y="20" width="100" height="100"
  style="fill: url(#reflected); stroke: black;"/>
```

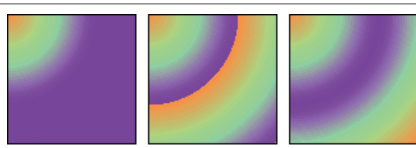


图 8-15：径向渐变 spreadMethod（值为 pad、repeat 和 reflect）的效果

8.2.3　渐变总结

线性渐变和径向渐变描述了使用平滑过渡的颜色填充对象。我们所讨论的对象都有一个边

界框，它们被定义为完全包含对象的最小矩形。<linearGradient> 和 <radialGradient> 元素都是一系列 <stop> 元素的容器。每个 <stop> 元素都指定 stop-color、offset 和可选的 stop-opacity。对于线性渐变，偏移就是渐变线性方向的距离的百分比。对于径向渐变，偏移就是渐变半径的距离的百分比。

线性渐变中，方向的起点（拥有 0% 渐变点）由属性 x1 和 y1 定义，终点（拥有 100% 渐变点）由属性 x2 和 y2 定义。

对于径向渐变，焦点（拥有 0% 渐变点）由属性 fx 和 fy 定义，拥有 100% 渐变点的圆由中心点坐标 cx、cy 和半径 r 定义。

如果 gradientUnits 属性值为 objectBoundingBox，则把边界框尺寸的百分比（默认行为）作为坐标。如果设置为 userSpaceOnUse，则采用填充对象的坐标系统。

如果线性渐变或者径向渐变的方向没有达到填充对象的边界，剩余空间的着色由 spreadMethod 属性值确定：默认值 pad 将起点和终点颜色扩展到边界，repeat 会重复起点到终点的渐变直到达到边界，reflect 以终点到起点、起点到终点的形式重复渐变，直到达到对象的边界。

8.3 变换图案和渐变

有时候可能需要斜切、拉伸或者旋转图案或者渐变。此时无需变换填充的对象，而是变换用来填充对象的图案或者渐变。可以用 gradientTransform 和 patternTransform 属性来实现，见示例 8-16，其结果如图 8-16 所示。

示例 8-16：变换图案和渐变

http://oreillymedia.github.io/svg-essentials-examples/ch08/pattern_gradient_transform.html

```
<defs>
  <pattern id="tile" x="0" y="0" width="20%" height="20%"
      patternUnits="objectBoundingBox">
  <path d="M 0 0 Q 5 20 10 10 T 20 20"
      style="stroke: black; fill: none;"/>
  <path d="M 0 0 h 20 v 20 h -20 z"
      style="stroke: gray; fill: none;"/>
  </pattern>

  <pattern id="skewed-tile"
    patternTransform="skewY(15)"
    xlink:href="#tile"/>

  <linearGradient id="plain">
    <stop offset="0%" style="stop-color: #ffcc00;"/>
    <stop offset="33.3%" style="stop-color: #cc6699;"/>
    <stop offset="100%" style="stop-color: #66cc99;"/>
```

```
  </linearGradient>

  <linearGradient id="skewed-gradient"
    gradientTransform="skewX(10)"
    xlink:href="#plain"/>
</defs>

<rect x="20" y="10" width="100" height="100"
    style="fill: url(#tile); stroke: black;"/>
<rect x="135" y="10" width="100" height="100"
    style="fill: url(#skewed-tile); stroke: black;"/>

<rect x="20" y="120" width="200" height="50"
    style="fill: url(#plain); stroke: black;"/>
<rect x="20" y="190" width="200" height="50"
    style="fill: url(#skewed-gradient); stroke: black;"/>
```

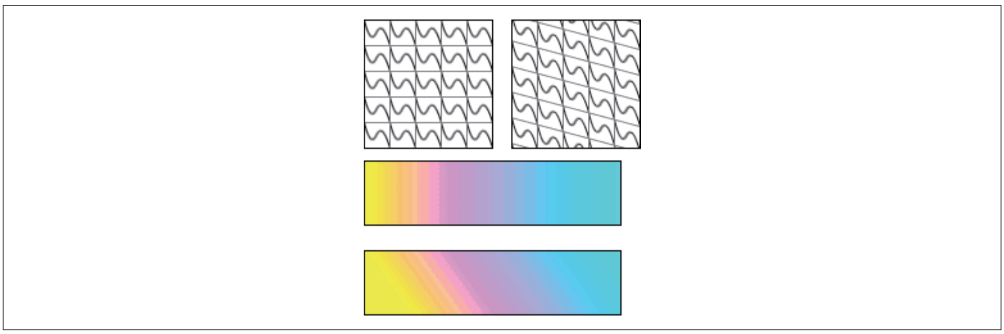


图 8-16：变换图案和渐变

关于渐变和图案的最后一个注意事项是，尽管这些例子都只将它们应用于形状的填充区域，但其实它们还可以用于 stroke。这样就可以为对象生成彩色或图案描边。为了让这种效果更清晰可见，通常要设置 stroke-width 为大于 1 的数字。

> 在添加 stroke 的时候，大小是基于 objectBoundingBox 来计算的。因为水平线和垂直线的边界框的宽度或高度默认为 0，所以当图案和渐变作用于这些线条的时候，使用 objectBoundingBox 单位的渐变或者图案会被忽略。这意味着它们完全不会被绘制出来，除非在样式（style）中指定备选笔画值，比如 stroke: url(#rainbow) red。
>
> 如果图案或渐变定义在独立的文件中，设置备用填充和笔画选项是个好主意，以防文件不能加载或者 SVG 阅读器不支持外部引用。

第 9 章

文本

如果说每张图片都在讲述一个故事的话，那么加上一些文字会更完美。SVG 中有好几个用于在图像中加入文本的元素。

9.1 文本的相关术语

在介绍 `<text>` 元素（添加文本的主要方法）之前，我们需要先定义一些术语。如果查看 SVG 的规范或者在任何图像系统中处理文本，都会看到这些术语。

- 字符

  在 XML 文档中，字符是指带有一个数字值的一个或多个字节，数字值与 Unicode 标准对应。例如，字母 g 是 Unicode 值为 103 的字符。

- 符号

  符号（glyph）是指字符的视觉呈现。每个字符都可以用很多不同的符号来呈现。图 9-1 展示了用两种不同的符号呈现的单词“glyphs”。注意开头的字母 g，同一个字符，但符号却明显不同。

glyphs　glyphs

图 9-1：两种符号

一个符号可能由多个字符构成。一些字体为特定的字母组合（如 fl 和 ff）准备了单独的符号，以使它们更好看，这种特性叫作“连字”（ligature）。有时候，一个字符也可能由几个

符号组合而成，比如打印程序可能会组合符号 e 和重音符号（´）来打印字符 é（Unicode 值为 233）。

- *字体*

  字体是指代表某个字符集合的一组符号。

一套字体中的所有符号一般都有以下特性。

- *基线、上坡度、下坡度*

  字体中的所有符号以基线对齐。基线到字体中最高字符顶部的距离称为上坡度（ascent），基线到最深字符底部的距离称为下坡度（descent）。字符的总高度为上坡度和下坡度之和，也称为 em 高度。em-box 是指宽度为 em 高度的方块。

- *大写字母高度、x高度*

  大写字母高度（cap-height）是指基线上的大写字母的高度；*x* 高度更好理解，是指基线到小写字母 *x* 顶部的高度。*x* 高度通常能比 em 高度更好地衡量一个字体的尺寸和可读性。

图 9-2 中标识了一套典型的罗马字母字体中的基线、上坡度、下坡度。上方的虚线表示大写字母高度，下面的虚线标记了 *x* 高度。

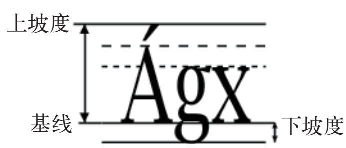


图 9-2：符号的度量

9.2 <text>元素的基本属性

使用 <text> 元素最简单的情况是只给定 x 和 y 属性的值，用来指定元素内容的第一个字符的基线位置。和所有对象一样，文本的默认样式是黑色填充、没有轮廓。这刚好是你想要的。如果同时设置轮廓和填充，那么文本会变粗，看着并不舒服。如果只设置轮廓样式，文本看起来就会很舒服，尤其是将笔画宽度设置得比较小的时候。示例 9-1 展示了 <text> 的位置属性和笔画 / 填充样式，结果见图 9-3。

示例 9-1：文本的位置和轮廓

```
<!-- 参考线 -->
<path d="M 20 10, 20 120 M 10 30 100 30 M 10 70 100 70
   M 10 110 100 110" style="stroke: gray;"/>

<text x="20" y="30">Simplest Text</text>
```

```
<text x="20" y="70" style="stroke: black;">Outlined/filled</text>
<text x="20" y="110" style="stroke: black; stroke-width: 0.5;
    fill: none;">Outlined only</text>
```



图 9-3：文本的位置和轮廓

还有很多可以应用到文本上的属性和 CSS 标准中是一样的。下面是一些在 Apache Batik 1.7 阅读器中实现的 CSS 属性和对应的值。它们也适用于大部分（但不是所有）阅读器。

- `font-family`

  值为由空格分隔的一系列字体名称或者通用字体名称。这些字体并不会全部生效，而是会按顺序依次回退（fallback），即 SVG 阅读器会按从前往后的顺序使用它识别出的第一个字体。通用字体必须放在最后。SVG 阅读器被要求必须能识别出通用字体名称，并且拥有可用字体。通用字体包括 `serif`、`sans-serif`、`monospace`、`fantasy`、`cursive`。`serif` 表示衬线字体，字体的边缘有小“勾”，而 `sans-serif`（非衬线字体）则没有。在图 9-1 中，左边的单词使用的是衬线字体，右边的单词使用的是非衬线字体。衬线字体和非衬线字体都是不等宽的，大写字母 M 和大写字母 I 的宽度是不同的。而 `monospace`（等宽字体）则不管有没有衬线，所有符号的宽度都一样，就像打字机打出来的那样。`fantasy` 和 `cursive` 代表的默认字体在不同的 SVG 阅读器上可能会有很大不同。

- `font-size`

  如果有多行文本的话，`font-size` 的值为相邻的两条基线的距离。（在 SVG 中，开发者必须自己定位多行文本，所以这个概念有点抽象。）如果指定了单位，比如 `style="font-size: 12pt"`，则在渲染前字体大小会被转换为用户坐标，所以它可能会受变换和 SVG `viewBox` 影响。如果你使用相对单位（`em`、`ex` 或者百分比），这些单位会相对于继承的字体大小进行计算。

- `font-weight`

  最常用的两个值为 `bold` 和 `normal`。当需要将一堆设置了 `style="font-weight: bold"` 的文本中的一部分变为非粗体时，就需要设置值为 `normal`。

- `font-style`

  最常用的两个值为 `italic`（斜体）和 `normal`。

- `text-decoration`

  可能的值为 `none`、`underline`（下划线）、`overline`（上划线）、`line-through`（删除线）。

- word-spacing

  该属性的值为一个长度，可以显式带上单位（如 pt），也可以使用用户坐标。正值将增大单词之间的间距，normal 将保持正常间距，负值会减小单词之间的间距。指定的值将与正常间距相加。

- letter-spacing

  该属性的值为一个长度，可以显式带上单位（如 pt），也可以使用用户坐标。正值将增大字母之间的间距，normal 将保持正常间距，负值会减小字母之间的间距。指定的值将与正常间距相加。

示例 9-2 使用了这些属性，结果见图 9-4，这些效果你可以在任何字处理软件中看到。

示例 9-2：文本样式

```
<g style="font-size: 18pt">
<text x="20" y="20" style="font-weight:bold;">bold</text>
<text x="120" y="20" style="font-style:italic;">italic</text>
<text x="20" y="60" style="text-decoration:underline;">under</text>
<text x="120" y="60" style="text-decoration:overline;">over</text>
<text x="200" y="60" style="text-decoration:line-through;">through</text>
<text x="20" y="90" style="word-spacing: 10pt;">more word space</text>
<text x="20" y="120" style="word-spacing: -3pt;">less word space</text>
<text x="20" y="150" style="letter-spacing: 5pt;">wide letter space</text>
<text x="20" y="180"
      style="letter-spacing: -6pt;">narrow letter space</text>
</g>
```

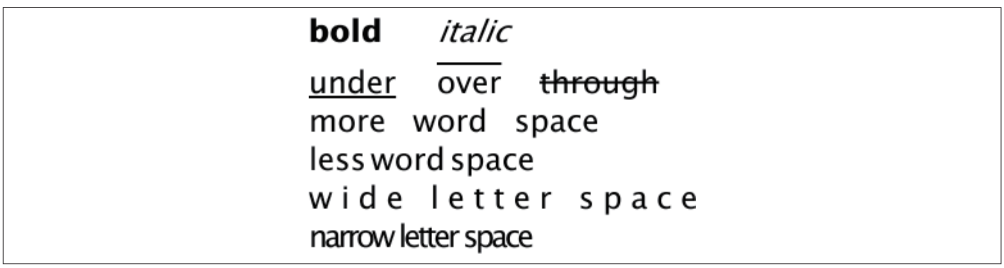


图 9-4：文本样式

9.3 文本对齐

<text> 元素让你指定了起始点，但是你并不能事先知道它的终点。这使得让文本居中对齐或者右对齐变得很困难。我们可以使用 text-anchor 属性来指定文本坐标生效的位置，它的值可以是 start、middle 或者 end。如果文字是从左向右书写的，这三个值分别表示左对齐、居中对齐和右对齐。如果文字的书写方向不是从左到右（见 9.7 节），则会有不同的效果。示例 9-3 展示了三个文本，它们的 *x* 坐标都是 100，但是 text-anchor 值不一样。为了看得更清晰，图上画了一条参考线，见图 9-5。

示例 9-3：text-anchor 的使用

http://oreillymedia.github.io/svg-essentials-examples/ch09/text_alignment.html

```
<g style="font-size: 14pt;">
  <path d="M 100 10 100 100" style="stroke: gray; fill: none;"/>
  <text x="100" y="30" style="text-anchor: start">Start</text>
  <text x="100" y="60" style="text-anchor: middle">Middle</text>
  <text x="100" y="90" style="text-anchor: end">End</text>
</g>
```

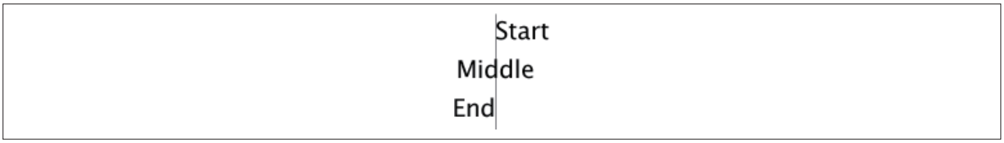


图 9-5：使用 text-anchor 的效果

9.4 <tspan>元素

无法提前预知文本元素的长度带来的另一个问题是，很难为一个字符串应用不同的文本属性，比如一个句子中穿插着斜体、正常字体和粗体。如果只使用 <text> 元素，则需要反复试验以确定将这些不同格式的字符串放在什么位置。为了解决这个问题，SVG 提供了 <tspan> 元素。与 (X)HTML 中的 <span> 元素类似，<tspan> 元素可以嵌套在文本内容中，并可以改变其中文本的样式。<tspan> 元素知道文本的位置，所以不需要你操心。示例 9-4 的效果如图 9-6。

示例 9-4：使用 <tspan> 改变文本样式

```
<text x="10" y="30" style="font-size:12pt;">
  Switch among
  <tspan style="font-style:italic">italic</tspan>, normal,
  and <tspan style="font-weight:bold">bold</tspan> text.
</text>
```

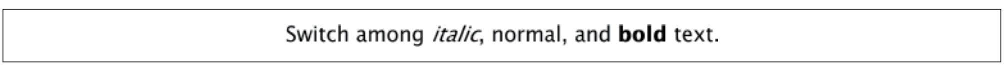


图 9-6：使用 <tspan> 改变文本样式

除了可以改变像字体大小、颜色、字重之类的表现样式外，还可以在 <tspan> 上应用一些属性来改变某个字母或者某些字母的位置。比如，如果你想应用上标或者下标样式，可以使用 dy 属性来改变字母的偏移量。这个属性值会被加到当前字符的垂直位置上，并持续生效，即使在 <tspan> 元素关闭后仍然有效。这个属性允许设置负值。另一个相似的属性是 dx，会改变字母在水平方向上的偏移量。示例 9-5 使用垂直偏移量来创建“掉落的字母”，效果见图 9-7。

示例 9-5：使用 dy 改变文本的垂直位置

```
<text x="10" y="30" style="font-size:12pt;">
  F <tspan dy="4">a</tspan>
  <tspan dy="8">l</tspan>
  <tspan dy="12">l</tspan>
</text>
```

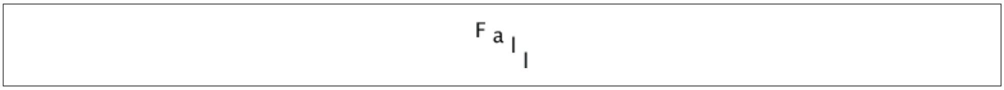


图 9-7：使用 dy 改变文本的垂直位置

如果你想使用绝对位置来设定偏移量，而不是相对元素本身的偏移量来设置，要使用 x 和 y 属性。在处理多行文本时使用这两个属性很方便。在 9.9 节中我们将看到，SVG 不会处理换行符，不会自动断行，所以你需要手动为每一行设置 x 属性值，并使用 y 或者 dy 来垂直定位。你应该始终在 <text> 元素中使用 <tspan>，以便将相关联的行进行分组，这样不仅可以将它们作为一个单位一起选中，也会使文档更加结构化。示例 9-6 展示了 Edward Lear 的 *The Owl and the Pussycat* 这首诗中的一节，使用了 <tspan> 元素，并使用了绝对 x 坐标和 y、dy 属性进行定位。

示例 9-6：使用 <tspan> 进行绝对定位

```
<text x="10" y="30" style="font-size:12pt;">
  They dined on mince, and slices of quince,
  <tspan x="20" y="50">Which they ate with a
    runcible spoon;</tspan>
  <tspan x="10" y="70">And hand in hand, on the edge
    of the sand,</tspan>
  <tspan x="20" dy="20">They danced by the light of the moon.</tspan>
</text>
```

图 9-8 中没有明显的证据证明所有的文本都在同一个 <text> 元素中，但请对我们的代码有信心，它们确实是在一起的。

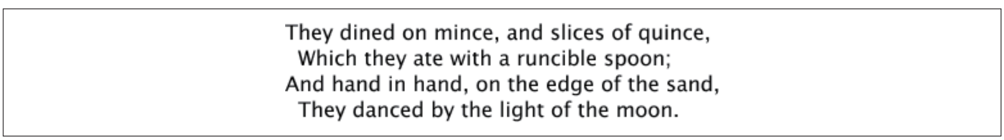


图 9-8：绝对定位的诗

你也可以使用 rotate 属性对 <tspan> 中的单个字母或者一些字母进行旋转，它的值是以度为单位的角度值。

如果你需要一次修改多个字母的位置，可以为 x、y、dx、dy 和 rotate 属性一次设置一系列的值。你指定的值将会按顺序一个一个应用到 <tspan> 中的字母上。见示例 9-7。

示例 9-7：在 <tspan> 中为 dx、dy、rotate 设置多个值

```
<text x="30" y="30" style="font-size:14px">It's
<tspan dx="0 4 -3 5 -4 6" dy="0 -3 7 3 -2 -8"
  rotate="5 10 -5 -20 0 15">shaken</tspan>,
not stirred.
</text>
```

注意在图 9-9 中，dx 和 dy 在 <tspan> 关闭后仍然有效，在 </tspan> 之后的文本的偏移量和 shaken 中的 n 是一样的。后面的文本并没有回到 <tspan> 中第一个字母的基线。

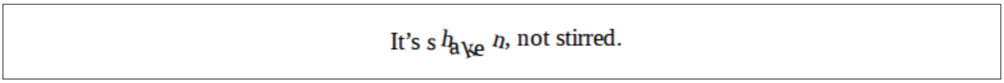


图 9-9：多个垂直和水平偏移量值

尽管可以使用 dy 属性来产生上标和下标，但使用 baseline-shift 属性会更简单，见示例 9-8。这个属性的值可以为 super 和 sub。你也可以指定一个长度值，如 0.5em，或者相对字体尺寸进行计算的百分比。baseline-shift 的影响范围仅限于它所在的 <tspan> 元素。

示例 9-8：baseline-shift 的使用

```
<text x="20" y="25" style="font-size: 12pt;">
C<tspan style="baseline-shift: sub;">12</tspan>
H<tspan style="baseline-shift: sub;">22</tspan>
O<tspan style="baseline-shift: sub;">11</tspan> (sugar)
</text>

<text x="20" y="70" style="font-size: 12pt;">
6.02 x 10<tspan baseline-shift="super">23</tspan>
(Avogadro's number)
</text>
```

图 9-10 中的上下标数字似乎有点大。设置一个 font-size 属性应该会更好，不过我们只希望用这个例子来说明一个问题。

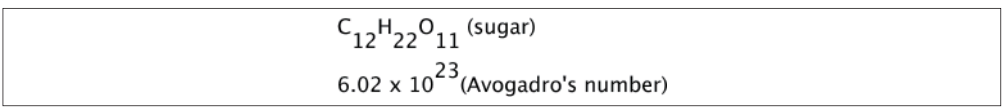


图 9-10：上下标

9.5 设置文本长度

尽管之前说过，我们无法提前知道一段文本的结束位置，但可以使用 textLength 属性显式设置文本的长度。SVG 会将文本调整到指定的长度。在调整的时候，可以只调整字符之间的间距，保持字符本身大小不变，也可以同时调整字符的间距和字符本身的大小。如果你只想调整字符的间距，可以将 lengthAdjust 属性值设为 spacing（默认值）。

如果你希望 SVG 同时调整字符间距和字符本身的大小，则将 lengthAdjust 值设置为 spacingAndGlyphs。示例 9-9 使用了这些属性，效果见图 9-11。

示例 9-9：textLength 和 lengthAdjust 的使用

http://oreillymedia.github.io/svg-essentials-examples/ch09/text_length.html

```
<g style="font-size: 14pt;">
<path d="M 20 10 20 70 M 220 10 220 70" style="stroke: gray;"/>
<text x="20" y="30"
  textLength="200" lengthAdjust="spacing">Two words</text>
<text x="20" y="60"
  textLength="200" lengthAdjust="spacingAndGlyphs">Two words</text>

<text x="20" y="90">Two words
  <tspan style="font-size: 10pt;">(normal length)</tspan></text>

<path d="M 20 100 20 170 M 100 100 100 170" style="stroke: gray;"/>
<text x="20" y="120"
  textLength="80" lengthAdjust="spacing">Two words</text>
<text x="20" y="160"
  textLength="80" lengthAdjust="spacingAndGlyphs">Two words</text>
</g>
```

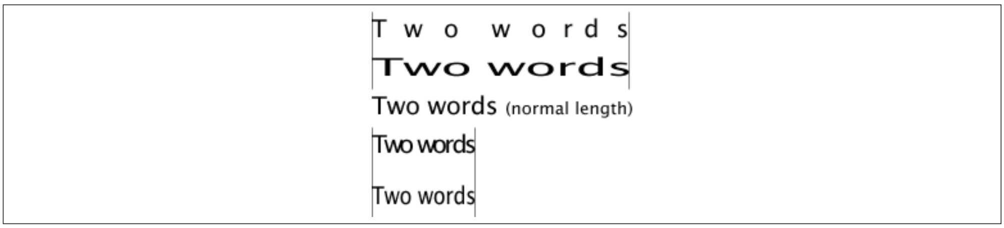


图 9-11：textLength 和 lengthAdjust 不同值的效果

9.6 纵向文本

当使用 SVG 创建报表、图形或者表格的时候，经常会希望文本标签能沿着垂直坐标轴排列。要实现这样的效果，一种方法是使用变换（transform）将文本旋转 90 度。另一种方法是将 writing-mode 属性值设为 tb（top to bottom，从上到下）。

有时候，也会希望文本垂直排列时字母本身仍然是横向显示。

示例 9-10 通过将 glyph-orientation-vertical 属性值设为 0 实现了这样的效果。（默认值为 90，即将纵向排列的文本旋转 90 度。）在图 9-12 中，这个属性值会使得文本间距显得有点不自然，可以为 letter-spacing 设置一个小的负值解决这个问题。

示例 9-10：纵向文本

```
<text x="10" y="20" transform="rotate(90,10,20)">Rotated 90</text>
```

```
<text x="50" y="20" style="writing-mode: tb;">Writing Mode tb</text>
<text x="90" y="20" style="writing-mode: tb;
  glyph-orientation-vertical: 0;">Vertical zero</text>
```

如果你实践了这些例子，可能会发现一些特性（baseline-shift、间距、纵向文本）在一些阅读器中支持得很差。所以最好在你希望支持的 SVG 阅读器中测试自己的图形。

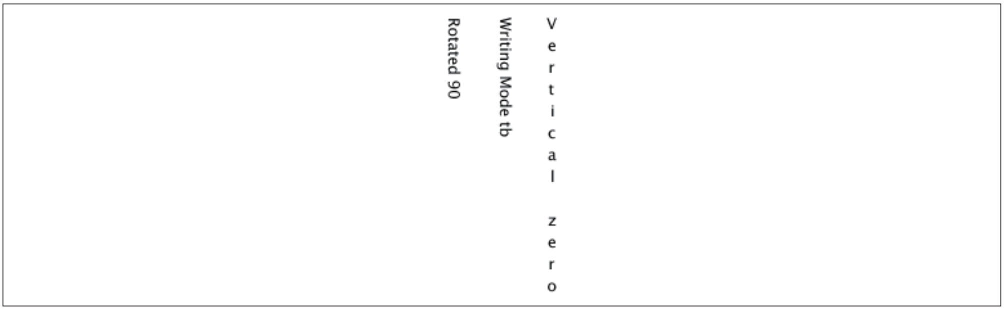


图 9-12：纵向文本

9.7 国际化和文本

如果你的图形中的文本需要被翻译为多种语言，那么 SVG 对 Unicode 的支持和在一个文档中显示各种语言的能力，能让你免去为每门语言创建一个文档的麻烦。

9.7.1 Unicode和双向语言

XML 是基于 Unicode 标准的（完整的文档见 Unicode 财团网站：http://www.unicode.org/）。这使得文本可以以阅读器软件支持的任何语言来显示，如图 9-13 所示。有一些语言，如阿拉伯语、希伯来语，书写方向是从右到左，所以当这些语言的文本与从左到右书写的语言（如英语）混在一起时，这些文本就是双向的（bidirectional），简称 bidi。系统软件知道每个字符的书写方向，并正确地计算出它们的位置。示例 9-11 中重设了文本的方向，将 direction 属性设为了 rtl，表示从右到左（right-to-left）。如果你想改变希伯来文或者阿拉伯文文本的方向，则需要将 direction 属性设为 ltr，表示从左到右（left-to-right）。你还需要将 unicode-bidi 属性值设为 bidi-override，来显式重设底层的 Unicode 双向文本算法。

示例 9-11：使用 Unicode 的国际化文本

```
<g style="font-size: 14pt;">

<text x="10" y="30">Greek: </text>

<text x="100" y="30">
  αβγδε
```

```
</text>

<text x="10" y="50">Russian:</text>
<text x="100" y="50">
  абвгд
</text>

<text x="10" y="70">Hebrew:</text>
<text x="100" y="70">
    אבגדה (written right to left)
</text>

<text x="10" y="90">Arabic:</text>
<text x="100" y="90">
  ا ب ج د (written right to left)
</text>

<text x="10" y="130">
  This is
    <tspan style="direction: rtl; unicode-bidi: bidi-override;
      font-weight: bold;">right-to-left</tspan>
  writing.
</text>
</g>
```

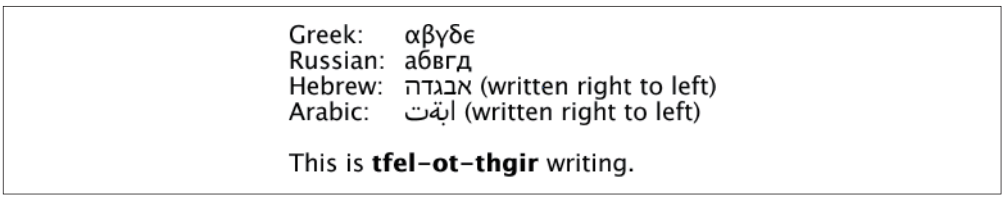


图 9-13：多种语言的文本

9.7.2 <switch>元素

在同一个文档中显示多种语言的能力在一些场景下很有用，比如创建一个迎接国际游客的小册子。有时候你会希望创建一个双语文档，比如西班牙语和俄语。使用西班牙语系统软件的人看到西班牙语文本，而使用俄语系统软件的人看到俄语文本。

SVG 通过 `<switch>` 元素提供了这种能力。这个元素会搜索所有的子节点，直到发现 `systemLanguage` 属性值与用户正在使用的软件的语言设置相符的节点。[1] `systemLanguage` 属性的值是一个语言名称或者使用逗号分隔的语言名称列表。语言名称要么是两个字母的语言代码，比如 `ru` 代表俄语，要么是语言代码加上国家代码，用于指定某个亚种语言，比如 `fr-CA` 代表加拿大法语，而 `fr-CH` 代表瑞士法语。

注 1：元素也可用于其他测试。2.3.1 节中介绍了如何用元素检测对某些特性的支持。如果在元素的子节点上使用多个测试属性，则它们全都必须匹配所要显示的内容。

一旦找到匹配的子节点，则这个节点所有的子节点都会被显示出来。<switch> 元素其他的子节点则会被忽略。示例 9-12 展示了以英式英语、美式英语、西班牙语和俄语显示的文本。如果语言代码能匹配上，则认为找到匹配节点，国家代码只是为了进一步确认，所以英式英语必须放在最前面。

示例 9-12：<switch> 元素的使用

```
<circle cx="40" cy="60" r="20" style="fill: none; stroke: black;"/>
<g font-size="12pt">
  <switch>
    <g systemLanguage="en-UK">
      <text x="10" y="30">A circle</text>
      <text x="10" y="100">without colour.</text>
    </g>
    <g systemLanguage="en">
      <text x="10" y="30">A circle</text>
      <text x="10" y="100">without color.</text>
    </g>
    <g systemLanguage="es">
      <text x="10" y="30">Un circulo</text>
      <text x="10" y="100">sin color.</text>
    </g>
    <g systemLanguage="ru">
      <text x="10" y="30">Kpyr</text>
      <text x="10" y="100">6e3 CBeTa.</text>
    </g>
  </switch>
</g>
```

图 9-14 是示例 9-12 在不同的语言设置下的截图。通常情况下应该再提供一个回退内容（在 <switch> 元素内容的最后，放置一个不写 systemLanguage 属性的分组）以便在匹配不到的时候显示。理想情况下，你应该为用户提供一种从可用语言中选择其一的方式。

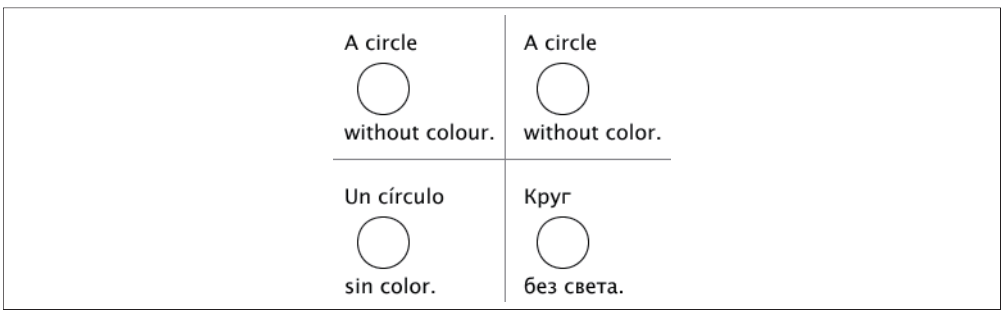


图 9-14：不同语言设置下的截图

9.7.3　使用自定义字体

有时候你需要一些 Unicode 中没有的特殊符号，或者希望只使用 Unicode 字符中的一部分，这

样就不用安装整个字体文件。如示例 9-15，仅仅需要 2000 多个韩文字符中的几个。你可以按照附录 E 制作一个自定义字体，然后为 <font> 起始标签设置一个唯一的 id。下面是包含从 Batang TrueType 字体中提取的 6 个韩文字符的字体文件的相关部分，文件名为 kfont.svg。

```
<font id="kfont-defn" horiz-adv-x="989" vert-adv-y="1200"
  vert-origin-y="0">
  <font-face font-family="bakbatn"
    units-per-em="1000"
    panose-1="2 3 6 0 0 1 1 1 1 1"
    ascent="800" descent="-200" baseline="0"/>
    <missing-glyph horiz-adv-x="500" />
    <!-- 字符定义在这里 -->
  </font-face>
</font>
```

서울 - 대한민국

图 9-15：使用外部字体的韩文字符

一旦字体文件提取完成后，示例 9-13 就可以从这个外部文件中引用字体。为了保持一致性，这个 SVG 文件中 font-family 的值应该与外部文件保持一致。

示例 9-13：外部字体的使用

```
<defs>
  <font-face font-family="bakbatn">
    <font-face-src>
      <font-face-uri xlink:href=:"kfont.svg#kfont-defn">
        <font-face-format string="svg" />
      </font-face-uri>
    </font-face-src>
  </font-face>
</defs>

<text font-size="28" x="20" y="40"
  style="font-family: bakbatn, serif;">
    서울-대한민국
</text>
```

SVG 字体目前不被 IE 浏览器支持（包括 IE11），也不被 Firefox 浏览器支持（版本 30）。在这些浏览器中，你可以再包含一个 <font-face-src> 元素，指定一个不同格式的字体 URI。或者，你可以使用只有一个 name 属性的 <font-face-name> 元素，来指定系统字体。这些元素与等价的 CSS font-face 属性有相同的写法。

如果你指定的字体全部不能使用，浏览器会尝试在系统字体中寻找一个能够显示文本中使用的字符的字体。

9.8　文本路径

文本并不一定要沿垂直或者水平的直线排列。它可以沿任何抽象路径排列，只需要简单地将文本放在 <textPath> 元素中，然后使用 xlink:href 属性引用一个之前已经定义好的 <path> 元素。字母会被旋转到与曲线垂直的方向“站立”（即基线是曲线的切线）。沿光滑连续曲线排列的文本比沿含有锐角或者不连续的路径排列的文本更易读。

> 在 <textPath> 元素中指定 <path> 并不会自动将路径显示出来。在示例 9-14 中，<path> 在 <defs> 中定义，它们通常不会被显示出来。示例中使用了 <use> 元素来显示这些线。

示例 9-14：textPath 示例

http://oreillymedia.github.io/svg-essentials-examples/ch09/text_path.html

```
<defs>
<path id="curvepath"
    d="M30 40 C 50 10, 70 10, 120 40 S 150 0, 200 40"
    style="stroke: gray; fill: none;"/>

<path id="round-corner"
    d="M250 30 L 300 30 A 30 30 0 0 1 330 60 L 330 110"
    style="stroke: gray; fill: none;"/>

<path id="sharp-corner"
    d="M 30 110 100 110 100 160"
    style="stroke: gray; fill: none;"/>

<path id="discontinuous"
    d="M 150 110 A 40 30 0 1 0 230 110 M 250 110 270 140"
    style="stroke: gray; fill: none;"/>
</defs>

<g style="font-family: 'Liberation Sans';
  font-size: 10pt;">
  <use xlink:href="#curvepath"/>
  <text>
    <textPath xlink:href="#curvepath">
    Following a cubic Bézier curve.
    </textPath>
  </text>

  <use xlink:href="#round-corner"/>
  <text>
    <textPath xlink:href="#round-corner">
    Going 'round the bend
    </textPath>
  </text>
```

```
    <use xlink:href="#sharp-corner"/>
    <text>
      <textPath xlink:href="#sharp-corner">
      Making a quick turn
      </textPath>
    </text>

    <use xlink:href="#discontinuous"/>
    <text>
        <textPath xlink:href="#discontinuous">
        Text along a broken path
        </textPath>
    </text>
  </g>
```

示例 9-14 的结果见图 9-16。图 9-17 展示了不将曲线画出来时文本的样子。

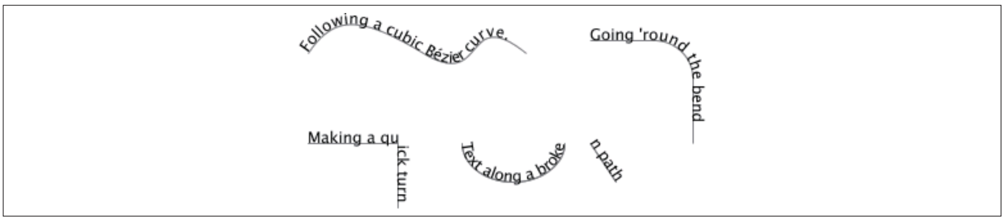


图 9-16：沿着路径排列的文本（画出路径）

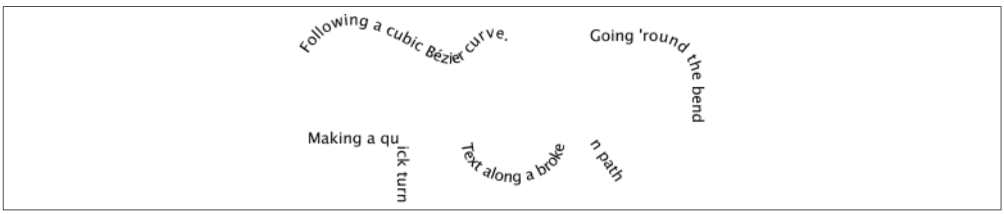


图 9-17：沿着路径排列的文本（未画出路径）

你可以通过设置 `startOffset` 属性（百分比或长度）来调整文本在路径上开始的位置。比如 `startOffset="25%"` 表示文本起始位置在路径的四分之一处，`startOffset="30"` 表示文本起始位置在路径开始 30 个用户单位后。如果你希望文本相对路径居中，如示例 9-15 中所示，只需要在 `<text>` 元素上设置 `textanchor="middle"` 并在 `<textPath>` 元素上设置 `startOffset="50%"` 即可。超出路径结尾处的文本会被截断，只会显示左边的部分，如图 9-18 所示。

示例 9-15：文本的长度和起始位置

http://oreillymedia.github.io/svg-essentials-examples/ch09/start_offset.html

```
  <defs>
    <path id="short-corner" transform="translate(40,40)"
```

```
      d="M0 0 L 30 0 A 30 30 0 0 1 60 30 L 60 60"
    style="stroke: gray; fill: none;"/>

  <path id="long-corner" transform="translate(140,40)"
      d="M0 0 L 50 0 A 30 30 0 0 1 80 30 L 80 80"
    style="stroke: gray; fill: none;"/>
</defs>

<g style="font-family: 'Liberation Sans'; font-size: 12pt">
  <use xlink:href="#short-corner"/>
  <text>
    <textPath xlink:href="#short-corner">
      This text is too long for the path.
      </textPath>
  </text>

  <use xlink:href="#long-corner"/>
  <text style="text-anchor: middle;">
    <textPath xlink:href="#long-corner" startOffset="50%">
      centered
    </textPath>
  </text>
</g>
```

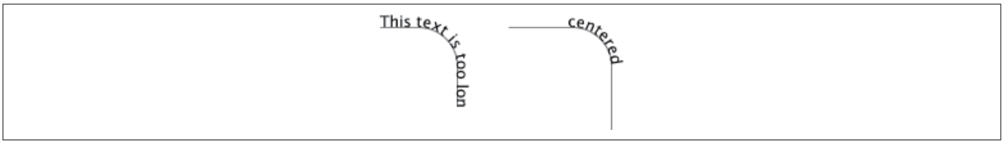


图 9-18：长文本和起始位置的效果

9.9 空白和文本

你可以通过改变 `xml:space` 属性的值来改变 SVG 处理文本中空白字符（空格、制表符、换行符）的方式。如果指定值为 `default`（默认值），则 SVG 会按如下规则处理空白字符：

- 删除所有换行符
- 将所有制表符转换为空格
- 删除首尾空格
- 将任意数量的连续空格换为一个空格

如果使用 `\t` 表示制表符，`\n` 表示换行符，下划线表示空格，则下面的字符串：

```
\n\n___abc_\t\t_def_\n\n__ghi
```

会渲染成：

```
abc_def_ghi
```

xml:space 的另一个值是 preserve，使用该值时，SVG 只会简单地将所有换行符和制表符换为空格（保留首尾空格），然后显示出来。同样的文本：

```
\n\n___abc_\t\t_def_\n\n__ghi
```

会渲染成这样：

```
_____abc____def_____ghi
```

> SVG 对空白字符的处理与 HTML 不完全一样。SVG 的默认处理方式会去掉所有的换行符，而 HTML 会将文本内容的换行符转换为空格。SVG 的 preserve 值将所有换行符转换为空格，而 HTML 的 <pre> 元素并不这么处理。在 SVG1.0 中无法产生新行，这使得人们感到很困惑，但 SVG 毕竟是面向图像显示的，而不是像 XHTML 那样面向文本内容，所以也可以理解。

9.10 案例学习：为图形添加文本

图 9-19 为韩国国旗添加了韩文和英文文本。文本相对椭圆路径居中。与示例 9-16 相比，新增的代码以粗体显示。

示例 9-16：文本案例学习

```
<defs>
  <font-face font-family="bakbatn">
    <font-face-src>
      <font-face-uri xlink:href="kfont.svg#kfont-defn">
        <font-face-format string="svg" />
      </font-face-uri>
    </font-face-src>
  </font-face>

  <path id="upper-curve" d="M -8 154 A 162 130 0 1 1 316 154"/>
  <path id="lower-curve" d="M -21 154 A 175 140 0 1 0 329 154"/>
</defs>

<ellipse cx="154" cy="154" rx="150" ry="120" style="fill: #999999;"/>
<ellipse cx="152" cy="152" rx="150" ry="120" style="fill: #cceeff;"/>

<!-- 浅红色大半圆填充符号的上半部分,其下方“浸入"符号左下方的浅红色小半圆 -->
<path d="M 302 152 A 150 120, 0, 1, 0, 2 152
  A 75 60, 0, 1, 0, 152 152" style="fill: #ffcccc;"/>

<!-- 浅蓝色小半圆,填充符号右上方 -->
<path d="M 152 152 A 75 60, 0, 1, 1, 302 152"
  style="fill: #cceeff;"/>

<text font-family="bakbatn, serif"
  style="font-size: 24pt; text-anchor: middle;">
```

```
    <textPath xlink:href="#upper-curve" startOffset="50%">
      서울 - 대한민국
    </textPath>
  </text>

  <text style="font-size: 14pt; text-anchor: middle;">
    <textPath xlink:href="#lower-curve" startOffset="50%">
    Seoul - Republic of Korea
    </textPath>
  </text>
```

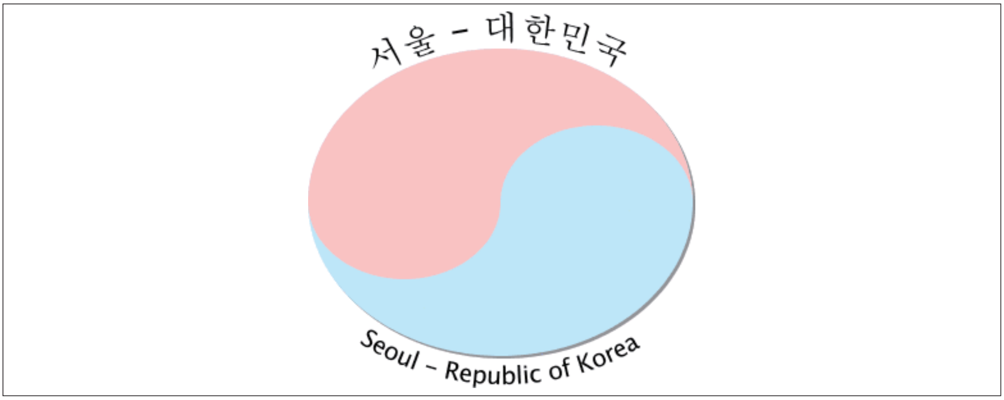


图 9-19：图形上添加的沿路径排列的文字

第 10 章

裁剪和蒙版

有时候我们可能不想看到完整的图片。例如，可能想要绘制一张像是通过双筒望远镜或者锁孔看到的图片，而镜筒和锁孔之外的所有部分都是不可见的。或者，想要绘制一张仿佛是通过半透明的幕布看到的图片。SVG 中使用裁剪和蒙版来实现这些效果。

10.1 裁剪路径

创建 SVG 文档时，可以通过指定感兴趣区域的宽度和高度建立视口。这会变成默认的裁剪区域，任何绘制在该范围外部的部分都不会显示。[1] 你还可以使用 `<clipPath>` 元素来建立自己的裁剪区域。

最简单的情形是建立一个矩形裁剪路径。在 `<clipPath>` 元素内是想要裁剪的 `<rect>`。因为我们只想要它的坐标，所以这个矩形本身不会显示。因此，可以在 `<clipPath>` 元素内随意指定填充和笔画风格。应用时在要裁剪的对象上添加 `clip-path` 样式属性，值引用到 `<clipPath>` 元素即可。注意这个属性带有连字符且不是大写的，裁剪元素则是有大写字母且不带连字符。示例 10-1 中，要裁剪的对象是第 1 章中的猫图像。结果如图 10-1 所示。

示例 10-1：裁剪矩形路径

http://oreillymedia.github.io/svg-essentials-examples/ch10/clip_path.html

```
<svg width="350" height="200" viewBox="0 0 350 200"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">
```

注 1：可以通过设置 `overflow` 样式属性为 `visible` 改变这一行为。

```
  <defs>
  <clipPath id="rectClip">
    <rect id="rect1" x="15" y="15"
      width="40" height="45"
      style="stroke: gray; fill: none;"/>
  </clipPath>
  </defs>

  <!-- 裁剪矩形 -->
  <use xlink:href="minicat.svg#cat"
    style="clip-path: url(#rectClip);"/>

  <!-- 引用图像,显示整个图像和裁剪区域轮廓 -->
  <g transform="translate(100,0)">
    <use xlink:href="#rect1"/>    <!-- 显示裁剪矩形 -->
    <use xlink:href="minicat.svg#cat"/>
  </g>
</svg>
```

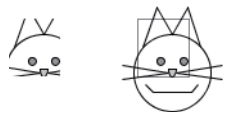


图 10-1：简单的矩形裁剪

<clipPath>，顾名思义，应该可以让我们指定任意形状的路径用于裁剪。事实上，<clipPath> 元素也确实可以包含任意数量的基本形状、<path> 元素或者 <text> 元素。示例 10-2 展示了一组按照曲线路径裁剪的形状和同一组根据文本裁剪的形状。

示例 10-2：复杂的裁剪路径

http://oreillymedia.github.io/svg-essentials-examples/ch10/complex_clip_path.html

```
<defs>
  <clipPath id="curveClip">
    <path id="curve1"
      d="M5 55 C 25 5, 45 -25, 75 55, 85 85, 20 105, 40 55 Z"
      style="stroke: black; fill: none;"/>
  </clipPath>

  <clipPath id="textClip">
    <text id="text1" x="20" y="20" transform="rotate(60)"
      style="font-family: 'Liberation Sans';
        font-size: 48pt; stroke: black; fill: none;">
       CLIP
     </text>
  </clipPath>

  <g id="shapes">
    <rect x="0" y="50" width="90" height="60" style="fill: #999;"/>
    <circle cx="25" cy="25" r="25" style="fill: #666;"/>
    <polygon points="30 0 80 0 80 100" style="fill: #ccc;"/>
```

```
  </g>
</defs>

<!-- 绘制曲线裁剪路径 -->
<use xlink:href="#shapes" style="clip-path: url(#curveClip);"/>

<!-- 绘制文本裁剪路径 -->
<use transform="translate(100, 0)"
  xlink:href="#shapes" style="clip-path: url(#textClip);"/>

<g transform="translate(0, 150)">
  <use xlink:href="#shapes"/>
  <use xlink:href="#curve1"/>   <!-- 显示裁剪路径 -->
</g>

<g transform="translate(100, 150)">
  <use xlink:href="#shapes"/>
  <use xlink:href="#text1"/>
</g>
```

为了帮助你更好地查看裁剪区域，前面的 SVG 在整个图像上方绘制了裁剪路径，在图 10-2 的右半部分可以看到。

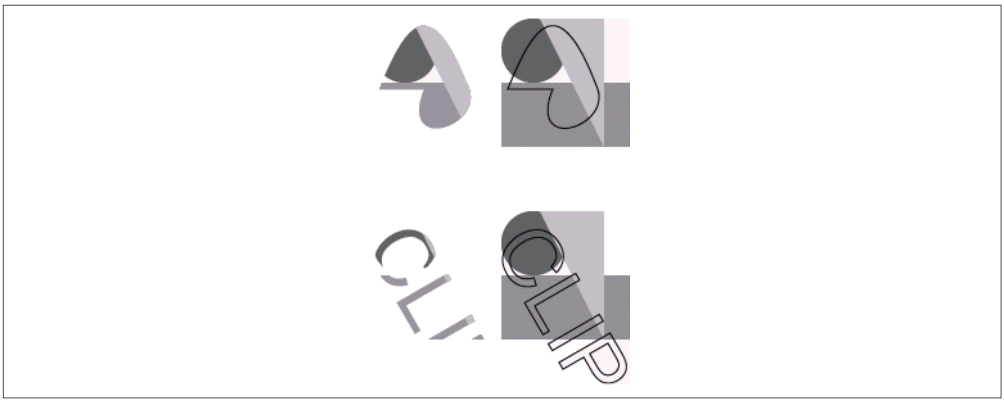


图 10-2：复杂的裁剪路径

前面的裁剪路径坐标被指定为用户坐标。如果想根据对象的边界框来表示坐标，设置 `clipPathUnits` 为 `objectBoundingBox` 即可（默认值为 `userSpaceOnUse`）。示例 10-3 使用路径裁剪在任意对象上生成一个圆形或者椭圆窗口。

示例 10-3：应用 `objectBoundingBox` 的 `clipPathUnits`

```
<defs>
 <clipPath id="circularPath" clipPathUnits="objectBoundingBox">
   <circle cx="0.5" cy="0.5" r="0.5"/>
 </clipPath>

  <g id="shapes">
```

```
      <rect x="0" y="50" width="100" height="50" style="fill: #999;"/>
      <circle cx="25" cy="25" r="25" style="fill: #666;"/>
      <polygon points="30 0 80 0 80 100" style="fill: #ccc;"/>
    </g>

    <g id="words">
      <text  x="0"  y="19" style="font-family: 'Liberation Sans';
        font-size: 14pt;">
      <tspan x="0"  y="19">If you have form'd a circle</tspan>
      <tspan x="12" y="35">to go into,</tspan>
      <tspan x="0"  y="51">Go into if yourself</tspan>
      <tspan x="12" y="67">and see how you would do.</tspan>
      <tspan x="50" y="87">&#8212;William Blake</tspan>
      </text>
    </g>
  </defs>

  <use xlink:href="#shapes" style="clip-path: url(#circularPath);"/>
  <use xlink:href="#words" transform="translate(110, 0)"
      style="clip-path: url(#circularPath);"/>
```

图 10-3 中，几何形状正好有个方形边界框，因此裁剪显示为圆形。文本由一个矩形区域限制，因此裁剪区域显示为一个椭圆。

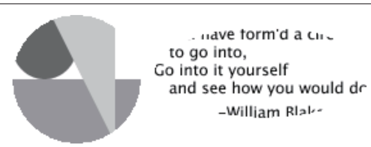


图 10-3：使用圆形 / 椭圆裁剪路径

<marker>、<symbol> 和 <svg> 元素都会定义其自身的视口，你也可以使用 overflow: hidden 样式来裁剪视口内容。然而，如果内容的 meet 设置为 preserveAspectRatio，视口可能比 viewBox 更大。要裁剪这个 viewBox，就要创建一个 <clipPath> 元素，其中包含一个匹配 viewBox 最小 x、最小 y、宽度和高度的矩形。

10.2　蒙版

SVG 中的蒙版与我们在化装舞会上戴的面具正好相反。在化装舞会上戴的面具，其不透明的部分用于遮挡我们的脸，半透明的部分让人能隐约看到我们的脸，而小孔（或透明部分）让人能清楚地看到我们的脸。而 SVG 蒙版会变换对象的透明度。如果蒙版是不透明的，被蒙版覆盖的对象的像素就是不透明的；如果蒙版是半透明的，那么对象就是半透明的，蒙版的透明部分会使被覆盖对象的相应部分不可见。

我们用 <mask> 元素创建蒙版。使用 x、y、width 和 height 属性指定蒙版的尺寸。这些尺

寸默认按照 objectBoundingBox 计算。如果想根据用户空间坐标计算尺寸，设置 maskUnits 为 userSpaceOnUse 即可。

起始 <mask> 和结束 </mask> 标记之间是我们想要用作蒙版的任意基本形状、文本、图像或者路径。这些元素的坐标默认使用用户坐标空间表达。如果想要为蒙版内容使用对象边界框，设置 maskContentUnits 为 objectBoundingBox 即可（默认为 userSpaceOnUse）。

接下来的问题是：SVG 如何确定蒙版的透明度，即阿尔法值？我们知道，每个像素由 4 个值描述，分别是红、绿、蓝颜色值和不透明度。虽然乍看之下仅使用不透明度值即可，但是 SVG 还是使用了所有可用的信息，而不是丢弃四分之三的像素信息。SVG 使用如下公式：

```
(0.2125 * red value + 0.7154 * green value + 0.0721 * blue value) * opacity value
```

所有的值都是 0 到 1 之间的浮点数。你可能对各个色值使用的系数不相同而感到惊讶，但是如果看看完全饱和的红色、绿色、蓝色，则会发现绿色似乎最亮，红色较暗，蓝色最暗（在图 10-4 中可看到）。颜色越暗，产生的阿尔法值越小，蒙版对象的不透明度越低。

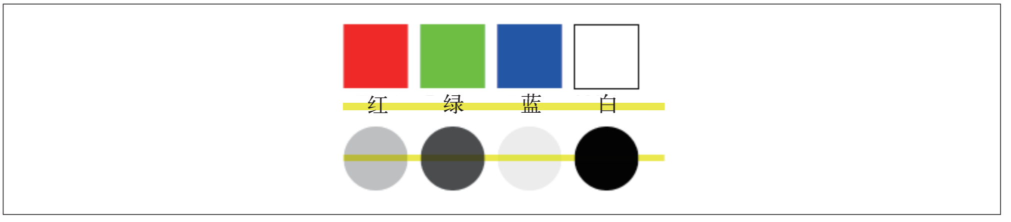


图 10-4：蒙版颜色透明度的效果

图 10-4 由示例 10-4 绘制，其中创建了黑色文本和黑色圆形，圆形由完全不透明的红、绿、蓝和白色正方形遮罩。文本和圆形被组合在一起，并且这个组合使用 mask 样式属性引用了对应的蒙版。背景中的水平黄色条演示了文本和圆形都是半透明的。

示例 10-4：不透明的颜色蒙版

```
<defs>
  <mask id="redmask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #f00;"/>
  </mask>

  <mask id="greenmask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #0f0;"/>
  </mask>

  <mask id="bluemask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #00f;"/>
```

```
  </mask>

  <mask id="whitemask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1" style="fill: #fff;"/>
  </mask>
</defs>

<!-- 显示颜色以演示相对亮度 -->
<rect x="10" y="10" width="50" height="50" style="fill: #f00;"/>
<rect x="70" y="10" width="50" height="50" style="fill: #0f0;"/>
<rect x="130" y="10" width="50" height="50" style="fill: #00f;"/>
<rect x="190" y="10" width="50" height="50"
    style="fill: #fff; stroke: black;"/>

<!-- 用于演示透明度的背景内容 -->
<rect x="10" y="72" width="250" height="5" style="fill: yellow;"/>
<rect x="10" y="112" width="250" height="5" style="fill: yellow;"/>

<g style="mask: url(#redmask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="35" cy="115" r="25"  style="fill: black;"/>
  <text x="35" y="80">Red</text>
</g>

<g style="mask: url(#greenmask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="95" cy="115" r="25" style="fill: black;"/>
  <text x="95" y="80">Green</text>
</g>

<g style="mask: url(#bluemask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="155" cy="115" r="25" style="fill: black;"/>
  <text x="155" y="80">Blue</text>
</g>

<g style="mask: url(#whitemask);
    font-size: 14pt; text-anchor: middle;">
  <circle cx="215" cy="115" r="25" style="fill: black;"/>
  <text x="215" y="80">White</text>
</g>
```

颜色、透明度和最终阿尔法值之间的关系并不直观。如果使用白色填充或者使用白色给遮罩内容描边，则上面公式中前半部分“颜色因子”结果为 1，此时不透明度则是控制蒙版阿尔法值的唯一因素。示例 10-5 就是使用这种方式编写的，其结果如图 10-5 所示。

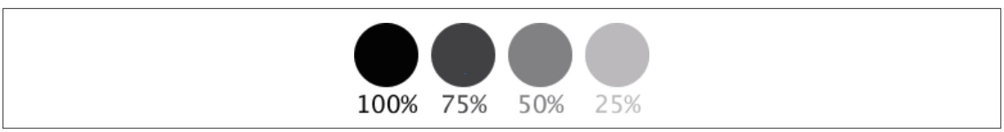


图 10-5：阿尔法值等于不透明度

示例 10-5：只使用不透明度的蒙版阿尔法值

http://oreillymedia.github.io/svg-essentials-examples/ch10/alpha_opacity_mask.html

```
<defs>
<mask id="fullmask" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 1.0; fill: white;"/>
</mask>

<mask id="three-fourths" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 0.75; fill: white;"/>
</mask>

<mask id="one-half" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 0.5; fill: white;"/>
</mask>

<mask id="one-fourth" x="0" y="0" width="1" height="1"
    maskContentUnits="objectBoundingBox">
    <rect x="0" y="0" width="1" height="1"
        style="fill-opacity: 0.25; fill: white;"/>
</mask>
</defs>

<g style="font-size: 14pt; text-anchor:middle; fill:black;">
    <g style="mask: url(#fullmask);">
    <circle cx="35" cy="35" r="25"/>
    <text x="35" y="80">100%</text>
    </g>

    <g style="mask: url(#three-fourths);">
    <circle cx="95" cy="35" r="25"/>
    <text x="95" y="80">75%</text>
    </g>

    <g style="mask: url(#one-half);">
    <circle cx="155" cy="35" r="25"/>
    <text x="155" y="80">50%</text>
    </g>

    <g style="mask: url(#one-fourth);">
    <circle cx="215" cy="35" r="25"/>
    <text x="215" y="80">25%</text>
    </g>
</g>
```

10.3　案例学习：为图形应用蒙版

示例 10-6 给 9.10 节中构造的图形添加了一个 JPG 图像。正如在图 10-6（缩小以节省空间）中所看到的，图像盖住了主椭圆内的曲线，并且蓝天也“侵入”了淡红色部分。

示例 10-6：图形内未应用蒙版的 <image>

```
<defs>
  <font-face font-family="bakbatn">
    <font-face-src>
      <font-face-uri xlink:href="kfont.svg#kfont-defn"/>
    </font-face-src>
  </font-face>
</defs>

<!-- 绘制椭圆和文本 -->
<use xlink:href="ksymbol.svg#ksymbol"/>

<image xlink:href="kwanghwamun.jpg" x="72" y="92"
    width="160" height="120"/>
```



图 10-6：图形内未应用蒙版的 <image> 的效果

解决方案是让照片的边缘淡出，使用径向渐变作为蒙版很容易做到。下面是要添加到文档 <defs> 部分的代码：

```
<radialGradient id="fade">
  <stop offset="0%" style="stop-color: white; stop-opacity: 1.0;"/>
  <stop offset="85%" style="stop-color: white; stop-opacity: 0.5;"/>
  <stop offset="100%" style="stop-color: white; stop-opacity: 0.0;"/>
</radialGradient>
<mask id="fademask" maskContentUnits="objectBoundingBox">
  <rect x="0" y="0" width="1" height="1"
    style="fill: url(#fade);"/>
</mask>
```

然后给 <image> 标记添加一个蒙版引用，结果如图 10-7 所示。

```
<image xlink:href="kwanghwamun.jpg" x="72" y="92"
  width="160" height="120"
  style="mask: url(#fademask);"/>
```



图 10-7：应用蒙版的图像

将图片的一部分隐藏起来可以大大改善整个图形的效果。

第 11 章 滤镜

前面的章节为我们提供了一些基础知识，以便创建传达精确、详细信息的图形。如果我们要去春游野餐，会想要一张精确的地图。当我们查看天气预报的图形时，想要的只是描述“事实”的部分。

如果稍后有人让你描述一下野餐那天的情况，他肯定不是想让你复述各种气象统计数据。同样，也没人想看一朵由纯矢量组成的春天的花；图 11-1 完全没有体现出温暖或魅力。



图 11-1：矢量组成的花朵

图形通常被设计用来激发人们的感情和情绪，也被用来传达信息。使用位图图形的艺术家们通常只关心对象的外观而不是它的几何定义，他们有很多工具可以用来添加这些效果。它们可以生成模糊的投影，加粗线条或者让线条变细，给绘图添加纹理，或者让对象看上去像浮雕或者倾斜的。

11.1 滤镜的工作原理

虽然 SVG 不是一种位图描述语言，但它仍然允许我们使用一些相同的工具。当 SVG 阅读

器程序处理一个图形对象时，它会将对象呈现在位图输出设备上；在某一时刻，阅读器程序会把对象的描述信息转换为一组对应的像素，然后呈现在输出设备上。例如我们用 SVG 的 <filter> 元素指定一组操作（也称作基元，primitive），在对象的旁边显示一个模糊的投影，然后把这个滤镜附加给一个对象：

```
<filter id="drop-shadow">
  <!-- 这是滤镜操作 -->
</filter>

<g id="spring-flower"
  style="filter: url(#drop-shadow);"/>
  <!-- 这里绘制花朵 -->
</g>
```

由于花朵在显示样式中用了滤镜，所以 SVG 不会将花朵直接渲染为最终图形。相反，SVG 会渲染花朵的像素到临时位图中。由滤镜指定的操作会被应用到该临时区域，其结果会被渲染为最终图形。

默认情况下临时位图的尺寸取决于渲染图像的显示屏的分辨率和尺寸。这就意味着即使所有的 SVG 代码是相同的，某些滤镜效果也可能在不同大小的显示屏中有不同的外观。规范中定义了一些属性来控制滤镜效果的有效分辨率（http://www.w3.org/TR/SVG11/filters.html#FilterEffectsRegion），但是它们在 SVG 阅读器中的实现并不一致，这里也不作讨论。

11.2　创建投影效果

示例 5-8 通过在着色椭圆下方偏移一个灰色椭圆，创建了一个投影效果。它是有效的，但是不优雅。让我们来研究一下如何使用滤镜创建更美观的投影。

11.2.1　建立滤镜的边界

<filter> 元素有一些属性用来描述该滤镜的裁剪区域。我们按照滤镜对象边界框的百分比指定 x、y、width 和 height（这也是默认方式）。任何在边界外部的输出都不会显示。如果想要为多个对象应用同一个滤镜，可能要完全忽略这些属性，并用默认值 x 等于 −10%，y 等于 −10%，width 等于 120%，height 等于 120%。这就为滤镜提供了额外的空间——这样构造的投影，产生的输出就会比输入大。

这些属性是按照滤镜对象的边界框来计算的，这是比较特殊的地方，即 filterUnits 的默认值是 objectBoundingBox。如果想要按照用户单位指定边界，设置这个属性的值为 userSpaceOnUse 即可。

还可以用 primitiveUnits 属性为用于滤镜基元中的单元指定单位。默认值为 userSpaceOnUse，但是如果设置为 objectBoundingBox，就可以按照图形尺寸的百分比来表示单位。

11.2.2　投影<feGaussianBlur>

起始和结束 <filter> 标记之间就是执行我们想要的操作的滤镜基元。每个基元有一个或多个输入，但只有一个输出。一个输入可以是原始图形（被指定为 SourceGraphic）、图形的阿尔法（不透明度）通道（被指定为 SourceAlpha），或者是前一个滤镜基元的输出。当只对图形的形状感兴趣而不管其颜色时，阿尔法源是有用的；它会避免阿尔法和颜色相互作用，正如 10.2 节中所描述的。

示例 11-1 第一次尝试为花朵生成一个投影，使用的是 <feGaussianBlur> 滤镜基元。我们指定 SourceAlpha 为它的输入源（用 in 属性），用 stdDeviation 属性指定它的模糊度。这个数值越大，模糊度越大。如果给 stdDeviation 值提供两个由空格分隔的数字，第一个数字被作为 *x* 方向的模糊度，而第二个被作为 *y* 方向的模糊度。

示例 11-1：生成投影的第一次尝试

```
<defs>
  <filter id="drop-shadow">
      <feGaussianBlur in="SourceAlpha" stdDeviation="2"/>
  </filter>
</defs>

<g id="flower" filter="url(#drop-shadow)">
    <!-- 在这里绘制花朵 -->
</g>
```

图 11-2 展示了结果，它可能不是我们所想的那样。

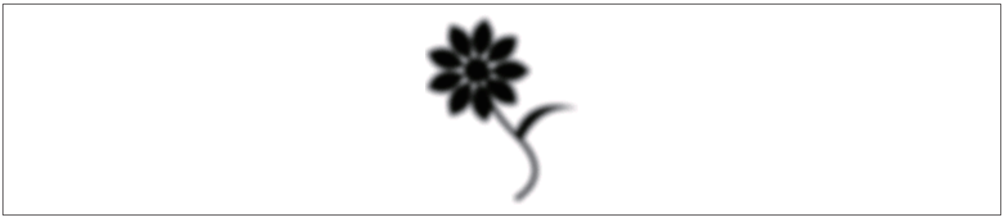


图 11-2：第一次尝试输出投影的结果

不要惊讶。记住，滤镜返回的输出是一个模糊的阿尔法通道，而不是原始图形。通过将花朵放入文档的 <defs> 中可以得到我们想要的效果，将 SVG 代码改变为如下所示：

```
<use xlink:href="#flower" filter="url(#drop-shadow)"
    transform="translate(4, 4)"/>
<use xlink:href="#flower"/>
```

然而，这就要求 SVG 对组成花朵的所有元素执行两次计算。更好的解决方案是添加更多的滤镜基元，让所有的工作可以在渲染期间一次完成。

11.2.3　存储、链接以及合并滤镜结果

示例 11-2 是改进后的滤镜。

示例 11-2：改进过的投影滤镜

```
<filter id="drop-shadow">
  <feGaussianBlur in="SourceAlpha" stdDeviation="2" result="blur">  ❶
  <feOffset in="blur" dx="4" dy="4" result="offsetBlur"> ❷
  <feMerge>    ❸
        <feMergeNode in="offsetBlur">
        <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
```

❶ `result` 属性指定当前基元的结果稍后可以通过 `blur` 名引用。这不同于 XML `id`；给定的名称是一个局部名称，它只在包含该基元的 `<filter>` 中有效。

❷ `<feOffset>` 基元接受它的输入，在这里就是 Gaussian blur 的返回结果 `blur`，它的偏移由 `dx` 和 `dy` 的值指定，然后将结果位图存储在 `offsetBlur` 名字下面。

❸ `<feMerge>` 基元包裹一个 `<feMergeNode>` 元素列表，其中每个元素都指定一个输入。这些输入按照它们出现的顺序一个堆叠在另一个上面。在这里我们希望 `offsetBlur` 在原始 `SourceGraphic` 下面。

现在，绘制花朵时就可以引用这个改进后的投影滤镜序列了，它会生成一个令人愉悦的如图 11-3 所示的图片。

```
<g id="flower" filter="url(#drop-shadow)">
    <!-- 这里绘制花朵 -->
</g>
```



图 11-3：改进后的投影结果

> 初次使用滤镜时，强烈建议你分阶段学习，一次测试一个滤镜。在尝试发现滤镜的工作原理期间，我就创建了大量丑陋的结果，你可能也会这样，就让它们成为我们之间的小秘密吧。
>
> 同样，初次学习滤镜时，可能会想在一个绘画上应用多个滤镜，看看会发生些什么。因为你的目的就是实验，所以尽管去做吧。一旦完成实验并开始生产工作，使用滤镜的目的就不一样了。滤镜应该用来帮助和增强我们的消息，而不是反过来让消息更不明显。明智地用一两个滤镜会有好处，而太多的滤镜几乎总是会弱化消息。

11.3　创建发光式投影

投影在花朵上的效果很好，但应用给文本时则完全不理想，正如在图 11-4 中看到的那样。



图 11-4：文本投影

如果文本周围有一个蓝绿色发光区域，看起来会更好看。可以用 `<feColorMatrix>` 基元来创建这个效果，使用它将黑色变为其他颜色。

11.3.1　<feColorMatrix>元素

`<feColorMatrix>` 元素允许我们以一种非常通用的方式改变颜色值。用于创建蓝绿色发光式投影的基元序列如示例 11-3 所示。

示例 11-3：发光滤镜

```
<filter id="glow">
  <feColorMatrix type="matrix"
    values=
      "0 0 0 0   0
       0 0 0 0.9 0
       0 0 0 0.9 0
       0 0 0 1   0"/>
  <feGaussianBlur stdDeviation="2.5"
    result="coloredBlur"/>
  <feMerge>
    <feMergeNode in="coloredBlur"/>
    <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
```

`<feColorMatrix>` 是一个通用的基元，允许我们修改任意像素点的颜色或阿尔法值。当 `type` 属性等于 `matrix` 的时候，我们必须设置 `value` 为 20 个数字来描述变换信息。这 20 个数字按照 4 行 5 列编写时最好理解。每行代表一个代数方程，定义了如何计算输出的 R、G、B、A 值（按行的顺序）。每行中的数字分别乘以输入像素的 R、G、B、A 的值和常量 1（按照列的顺序），然后加在一起得到输出值。要设置一个变换，将所有不透明区域绘制为相同的颜色，可以忽略输入颜色和常量，只要设置阿尔法列的值即可。这个矩阵模型看起来如下所示：

```
values=
  "0 0 0 red 0
   0 0 0 green 0
   0 0 0 blue 0
   0 0 0 1 0"
```

这里 red、green 和 blue 的值通常是 0 到 1 之间的十进制数。示例 11-3 中，red 设置为 0，green 和 blue 的值设置为 0.9，这会生成一个明亮的青色。

注意，这个例子中并没有一个用作这个基元输入的 `in` 属性，默认使用 `SourceGraphic`。这个基元上也没有 `result` 属性。这意味着这个颜色矩阵操作的输出只用于下一个滤镜基元的隐性输入。如果使用这种快捷方式，那么下一个滤镜基元一定不能有 `in` 属性。

在这个例子中，`<feColorMatrix>` 的结果是青色的色源。该滤镜的其他部分在它的基础上使用了一个高斯模糊；青色模糊结果被存储下来，以便将来用 `coloredBlur` 引用它。最后，`<feMerge>` 在对象下面产生输出发光。

我们可以使用这两个滤镜创建新的图形，改进后的 SVG 如下所示，结果见图 11-5。

```
<g id="flower" style="filter: url(#drop-shadow);">
  <!-- 绘制花朵 -->
</g>
<text x="120" y="50"
  style="filter: url(#glow); fill: #003333; font-size: 18;">
Spring <tspan x="120" y="70">Flower</tspan>
</text>
```



图 11-5：投影和发光文本

11.3.2 <feColorMatrix>详解

前面的例子使用的是最通用的一种颜色矩阵，可以任意指定我们想要的值。`type` 属性还有其他三个值。每个“内置”的颜色矩形都完成一个特定的视觉任务，并且都有自己的指定 `values` 的方式。

- `hueRotate`（色相旋转）

  `values` 是一个单一的数字，描述颜色的色相值应该被旋转多少度。完成这一任务的数学运算与 6.5 节中所描述的 `rotate` 变换非常相似。旋转和结果颜色之间的关系一点也不明显 [1]，如图 11-6 所示。你可以在在线版本中进行试验：

注 1：在 HSL 和 HSV 色彩空间中，H 指的就是色相，可以将不同色相的颜色分布在一个环上，以角度表示颜色。例如红色为 0 度（360 度），绿色为 120 度，蓝色为 240 度。可参见 http://zh.wikipedia.org/wiki/%E8%89%B2%E7%9B%B8。——译者注

http://oreillymedia.github.io/svg-essentials-examples/ch11/hue_rotate.html

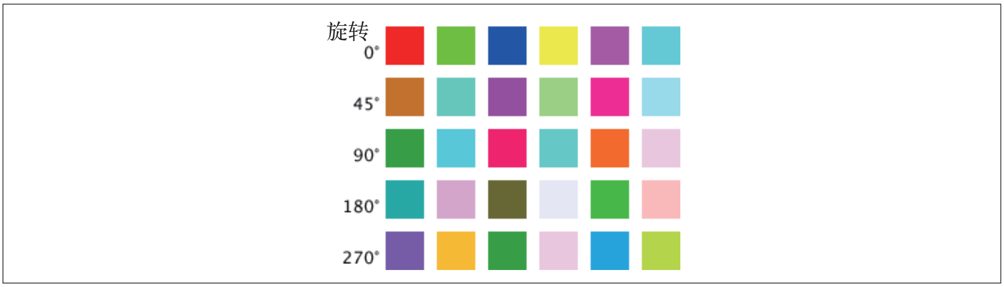


图 11-6：`hueRotate` 在完全饱和颜色上的效果

- `saturate`（饱和度）

  `values` 属性指定一个 0 到 1 之间的数字。数字越小，颜色“褪色”越多，正如在图 11-7 中可以看到的。值为 0，则将图形转换为黑白图。这个滤镜只可以用来降低图像的饱和度（洗白图像），不能增加饱和度（将正常图片变成彩色图像）。

  在线示例中可以测试其他饱和度值：

  http://oreillymedia.github.io/svg-essentials-examples/ch11/saturate.html



图 11-7：一些主要颜色不同饱和度下的效果

- `luminanceToAlpha`（用亮度决定阿尔法值）

  这个滤镜根据颜色的亮度建立阿尔法通道。这个亮度是颜色固有的“亮度”，正如 10.2 节中所描述的。图 11-8 中，颜色方块的亮度被用作实心黑方块的阿尔法通道。这个滤镜会丢弃原始方块的颜色，结果是具有不同透明度的纯黑色。颜色越浅，赋予滤镜对象的透明度越低。这一类型（`type`）忽略了 `values` 属性。

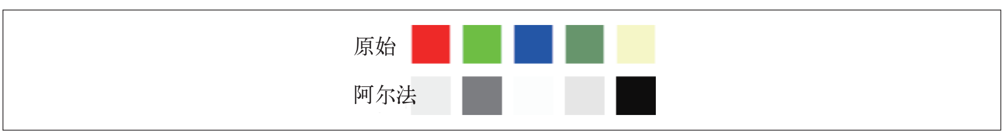


图 11-8：`luminanceToAlpha` 效果

11.4 <feImage>滤镜

到目前为止，我们只见过将原始图形或者阿尔法通道作为滤镜的输入源。SVG 的 <feImage> 元素允许我们使用任意的 JPG、PNG、SVG 文件，或者带有 id 属性的 SVG 元素作为滤镜的输入源。示例 11-4 导入了一个蓝天白云的图片用作花朵图片的背景。

示例 11-4：使用 <feImage> 元素

```
<defs>
<filter id="sky-shadow" filterUnits="objectBoundingBox">
    <feImage xlink:href="sky.jpg" result="sky"
         x="0" y="0" width="100%" height="100%"
         preserveAspectRatio="none"/>
    <feGaussianBlur in="SourceAlpha" stdDeviation="2" result="blur"/>
    <feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
    <feMerge>
        <feMergeNode in="sky"/>
        <feMergeNode in="offsetBlur"/>
        <feMergeNode in="SourceGraphic"/>
    </feMerge>
</filter>
</defs>

<g id="flower" style="filter: url(#sky-shadow)">
    <!-- 这里绘制花朵图形 -->
</g>

<!-- 展示原始图像 -->
<image xlink:href="sky.jpg" x="170" y="10"
    width="122" height="104"/>
```

图 11-9 展示了结果，原始的蓝天图片按照它本身的尺寸显示在右侧。图像默认被拉伸以适应定义在 <filter> 元素上的滤镜区域（默认情况下滤镜没有尺寸，因此默认滤镜区域是对象边界框超出 10% 的范围）。在 <feImage> 元素上可以设置明确的宽度、高度和 x/y 偏移。默认情况下，这些都使用 userSpaceOnUse 单位；然而，所有的百分比值都是相对于滤镜区域进行计算的。可以在 <filter> 元素上使用 primitiveUnits 属性来切换到 objectBoundingBox 的单位，但是这会影响滤镜中的所有元素。

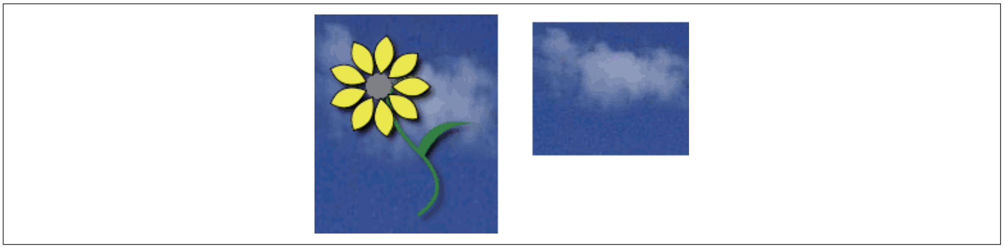


图 11-9：<feImage> 效果

除了导入一个完整的图像文件，还可以在 xlink:href 属性中使用 URL 片段，从某个文件中或者同一 SVG 文件的其他位置导入 SVG 图形的一部分到滤镜中。不幸的是，Mozilla FireFox 目前还不支持在 <feImage> 中使用图像片段。可以参考 Bugzilla bug 455986（https://bugzilla.mozilla.org/show_bug.cgi?id=455986）查看它的最新情况。

11.5 <feComponentTransfer>滤镜

上图中背景的问题在于它太暗了。调整 saturate 并不能达到预期的效果，它虽然改变了颜色的饱和度，但并不会改变亮度。要让图片变亮，需要增加每个颜色分量的值。虽然可以使用自定义颜色矩阵来达到这一效果，但是 <feComponentTransfer> 提供了一种更方便、更灵活的方式来单独操作每个颜色分量。它还允许我们对每个颜色分量作出不同的调整，因此我们既可以让蓝天更亮，也可以通过增加绿色和红色级别（多于蓝色级别），让它没有那么强烈。

可以通过在 <feComponentTransfer> 内配置 <feFuncR>、<feFuncG>、<feFuncB> 和 <feFuncA> 元素，调整红、绿、蓝色和阿尔法的级别。每个子元素都可以单独指定一个 type 属性，说明应如何修改该通道。

为了模拟亮度控制的效果，我们要指定 linear 函数，它会把当前颜色分量值 C 放到公式 slope * C + intercept 中。intercept 为结果提供了一个“基准值”，slope 是一个简单的比例因子。示例 11-5 用滤镜给带有投影的花朵添加了一个明亮的天空。注意，红色和绿色通道都做了与蓝色通道不同的调整。调整后天空变亮了很多，如图 11-10 所示。

示例 11-5：使用 <feComponentTransfer> 改变亮度

http://oreillymedia.github.io/svg-essentials-examples/ch11/linear_transfer.html

```
<filter id="brightness-shadow" filterUnits="objectBoundingBox">
  <feImage xlink:href="sky.jpg" result="sky"/>
  <feComponentTransfer in="sky" result="sky">
    <feFuncB type="linear" slope="3" intercept="0"/>
    <feFuncR type="linear" slope="1.5" intercept="0.2"/>
    <feFuncG type="linear" slope="1.5" intercept="0.2"/>
  </feComponentTransfer>
  <feGaussianBlur in="SourceAlpha" stdDeviation="2" result="blur"/>
  <feOffset in="blur" dx="4" dy="4" result="offsetBlur"/>
  <feMerge>
    <feMergeNode in="sky"/>
    <feMergeNode in="offsetBlur"/>
    <feMergeNode in="SourceGraphic"/>
  </feMerge>
</filter>
```



图 11-10：线性 <feComponentTransfer> 效果

简单的线性调整会为一个颜色分量中的所有值加上和乘以一个相同的量。但是 gamma 函数不是这样的，它把当前颜色值 C 放入了公式 amplitude * C^exponent + offset 中。offset 为结果提供了一个“基准值”，amplitude 是一个简单的比例因子，exponent 让结果与原始值的对应关系是一条曲线而不是直线。由于颜色值始终是 0 到 1 之间的数字，所以指数越大，修改后的值越小。图 11-11 展示了由指数值等于 0.6（黑色实线）、0.3（虚线）和 1.666 7（灰线）生成的曲线。

看看虚线，会看到较小的原始颜色值 0.1 会被提高到 0.5，增长了 400%。而原始值 0.5 只增加了 60% 到 0.8。它的作用是亮化整个图片，增加暗色区域的对比度，同时降低亮色区域的对比度。对于大于 1 的指数（灰线），修改后的值反而小于原始值，暗化图像的同时增加了明亮区域的对比度。注意，灰实线和黑实线相对于对角线是对称的：伽马值 1.6667 是伽马值 0.6 的逆矩阵。任何情况下，原始值为 0 或 1 时，指数都是没有作用的。

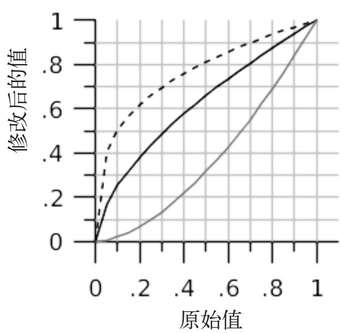


图 11-11：伽马曲线函数

当指定伽马滤镜时，我们设置 amplitude、exponent 和 offset 属性为前面公式中对应的值。示例 11-6 使用伽马校正调整了天空。在这个示例中，图 11-10 和图 11-12 之间的差异是较小的，但是有些图片通过这种方法会比前一种方法得到更多的改善。

示例 11-6：使用 <feComponentTransfer> 的伽马校正

http://oreillymedia.github.io/svg-essentials-examples/ch11/gamma_transfer.html

```
<feImage xlink:href="sky.jpg" result="sky"/>
<feComponentTransfer in="sky" result="sky">
  <feFuncB type="gamma"
    amplitude="1" exponent="0.2" offset="0"/>
  <feFuncR type="gamma"
    amplitude="1" exponent="0.707" offset="0"/>
  <feFuncG type="gamma"
    amplitude="1" exponent="0.707" offset="0"/>
</feComponentTransfer>
```



图 11-12：伽马校正的结果



精明的读者（就是你）可能会注意到，线性和伽马函数都可以生成大于 1.0 的颜色值。SVG 规范中说这并不是一个错误；在每个滤镜基元之后，SVG 处理程序都会将值固定在一个有效的范围内。因此，所有大于 1.0 的值都会被减小为 1.0，任何小于 0 的值都会被调整为 0。

<feComponentTransfer> 的 type 属性还有其他选项。注意，我们可以任意混合和匹配这些选项；可以针对红色值使用伽马校正，而用线性函数来亮化绿色值。

- identity

  一个“什么都不做”的函数。它允许我们明确规定颜色通道应该不受影响（如果不给通道提供一个 <feFuncX> 元素的话，这是默认行为）。

- table

  允许我们将颜色值划分为一系列相等的间隔，每个间隙中的值都相应地扩大。类似这样：最小的四分之一颜色范围的值加倍，下一个四分之一都塞入一个十分之一的范围，保持第三个四分之一的范围不变，然后将最后一个四分之一的值塞入剩下的 15% 的颜色范围中。

原始值范围	修改后的值范围
0.00~0.25	0.00~0.50
0.25~0.50	0.50~0.60
0.50~0.75	0.60~0.85
0.75~1.00	0.85~1.00

我们可以通过在 tableValues 属性中列出重映射范围的端点，指定绿色通道的映射：

```
<feFuncG type="table"
  tableValues ="0.0, 0.5, 0.6, 0.85, 1.0"/>
```

如果将输入范围划分为 *n* 个不同的部分，必须在 tableValues 中提供 *n*+1 个选项，使用空格或逗号分隔。

- discrete

  允许我们将颜色值划分为一系列相等的间隔，然后将每个都映射到一个离散的颜色值。类似这样：最低的四分之一颜色范围的值映射到 0.125，下一个四分之一映射到 0.375，第三个四分之一映射到 0.625，剩下的四分之一映射到 0.875（也就是每个四分之一范围都被映射到它们的中间值）。

原始值范围	修改后的值
0.00~0.25	0.125
0.25~0.50	0.375
0.50~0.75	0.625
0.75~1.00	0.875

我们可以在 tableValues 属性中列出离散值，指定绿色通道的映射，值之间用逗号或空格分隔：

```
<feFuncG type="discrete"
  tableValues ="0.125 0.375 0.625 0.875"/>
```

在 tableValues 属性中分割输入通道为几个部分就需要几个入口。例外情况：如果想要重新映射所有的输入值给单个输出值，必须将该入口放入 tableValues 中两次。因此，要设置蓝色通道的输入值为 0.5，就要这样：

```
<feFuncB type="discrete" tableValues="0.5 0.5"/>
```

如果想要反转通道的颜色值范围（即将从小到大的递增改变为从大到小的递减），使用这种方式：

```
<feFuncX type="table"
  tableValues="maximum minimum"/>
```

图 11-13 展示了使用 discrete 转换、table 转换以及反转 table 转换的效果。

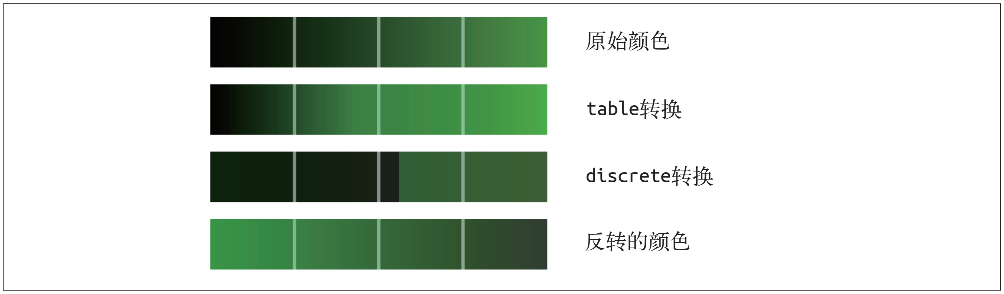


图 11-13：使用 table 和 discrete 转换的效果

定义色彩空间

通常，红、绿、蓝颜色分量的值被表示在一条从 0 到 1 的直线上，0 表示没有颜色，1 表示 100% 的颜色。这被称作线性颜色空间。然而，当 SVG 计算渐变点之间的颜色值时（正如 8.2 节中所描述的），SVG 使用一种特殊的方式表示颜色，而这样的值并不是一条 0 到 1 的直线。这种表示法被称作标准 RGB，也叫 sRGB 色彩空间（http://www.w3.org/Graphics/Color/sRGB.html），使用它可以使渐变的颜色更自然。图 11-4 展示了对比结果。第一个渐变是从黑色到绿色，第二个是从红色到绿色再到蓝色，而第三个是从黑色到白色。

默认情况下，滤镜算法会使用线性 RGB 空间的值计算所有插值颜色，因此如果我们给一个填充了渐变的对象应用滤镜，得到的结果可能根本就不是我们所期望的。为了得到正确的结果，我们必须通过给 `<filter>` 元素添加 `color-interpolation-filters="sRGB"` 属性，让滤镜按照 sRGB 色彩空间来计算。作为另一种选择，我们还可以选择不修改滤镜，而是给 `<gradient>` 元素应用 `color-interpolation="linearRGB"`，以让它使用的色彩空间与滤镜的默认色彩空间相同。

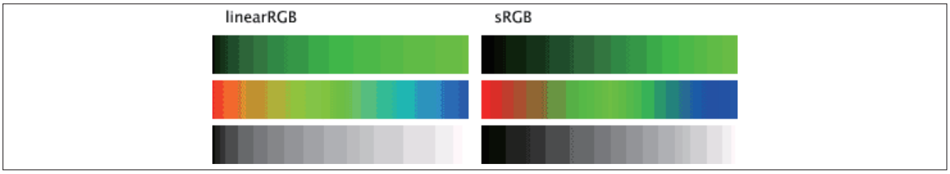


图 11-14：linearRGB 和 sRGB 对比

11.6 `<feComposite>`滤镜

到目前为止，我们已经通过使用 `<feMerge>` 将多个滤镜层叠起来，合并了多个滤镜的结果。而更常用的 `<feComposite>` 元素接受两个输入源，分别指定在 `in` 和 `in2` 属性中，它的 `operator` 属性用于设置如何合并这两个输入源。在下面的解释中，假设我们已经给前一个滤镜基元输出指定了 `result="A"` 和 `result="B"`：

- `<feComposite operator="over" in="A" in2="B"/>`

  生成的结果是 A 层叠在 B 上面，正如 <feMergeNode> 做的那样。事实上，<feMergeNode> 仅仅是指定 over 操作的 <feComposite> 元素的一种便利的快捷方式（<feMergeNode> 也允许我们一次层叠两个以上的图形）。

- `<feComposite operator="in" in="A" in2="B"/>`

  结果是 A 的一部分重叠在 B 的不透明区域。类似于蒙版效果，但是这个蒙版仅仅基于 B 的阿尔法通道，而不是它是颜色亮度。不要混淆这个属性值的名字和 in 属性。

- `<feComposite operator="out" in="A" in2="B"/>`

  结果是 A 的一部分位于 B 的不透明区域的外部（半透明区域有反转蒙版的效果）。

- `<feComposite operator="atop" in="A" in2="B"/>`

  结果是 A 的一部分位于 B 里面，B 的一部分在 A 外面。引用一下首先定义这些操作符的文章："……'报纸 atop 桌面'会包括报纸在桌面上的部分，以及桌面部分。在桌面之外的报表不在最终图形中。"（... paper atop table includes paper where it is on top of table, and table otherwise; area beyond the edge of the table is out of the picture.）[2]

- `<feComposite operator="xor" in="A" in2="B"/>`

  结果包含位于 B 的外面的 A 的部分和位于 A 的外面的 B 的部分。

- `<feComposite in="A" in2="B" operator="arithmetic".../>`

  灵活性最大。我们要提供 4 个系数：k1、k2、k3、k4。每个像素的每个通道的结果按照如下方式计算：

  ```
  k1 * A * B + k2 * A + k3 * B + k4
  ```

  这里的 A 和 B 是来自输入图形像素的颜色分量。

算术操作符在处理"溶解"效果时很有用。如果想要一个由图像 A 的 a% 和图像 B 的 b% 生成的结果图像，设置 k1 和 k4 为 0，k2 为 a/100 以及 k3 为 b/100 即可。例如，要创建一个 30% A 和 70% B 的混合效果，这样做就行了：

```
<feComposite in="A" in2="B" result="combined"
  k1="0" k2="0.30" k3="0.70" k4="0"/>
```

图 11-15 展示了我们所描述的组合，红色的文本是 in 图像，模糊偏移的投影是 in2 图像；arithmetic 参数为 50% 的文本和 50% 的投影。

注 2："合成数字图像"（Compositing Digital Images），T. Porter 和 T. Duff 著，计算机协会 SIGGRAPH 84 会议录，18 卷 3 号，1984 年 7 月。



图 11-15：使用 <feComposite> 运算符的效果

示例 11-7 用 in 和 out 运算符做了一个“镂空”效果。这个例子中移除了投影效果，最终生成如图 11-16 所示的视觉上更赏心悦目的效果。

示例 11-7：使用 <feComposite> in 和 out

```
<defs>
<filter id="sky-in" filterUnits="objectBoundingBox">
  <feImage xlink:href="sky.jpg" result="sky"
    x="0" y="0" width="100%" height="100%"
    preserveAspectRatio="none"/>
  <feComposite in="sky" in2="SourceGraphic"
    operator="in"/>
</filter>

<filter id="sky-out" filterUnits="objectBoundingBox">
  <feImage xlink:href="sky.jpg" result="sky"
    x="0" y="0" width="100%" height="100%"
    preserveAspectRatio="none"/>
  <feComposite in="sky" in2="SourceGraphic"
    operator="out"/>
</filter>

<g id="flower">
  <!-- 这里绘制花朵图形 -->
</g>
</defs>

<use xlink:href="#flower" transform="translate(10,10)"
  style="filter: url(#sky-in);"/>

<use xlink:href="#flower" transform="translate(170,10)"
  style="filter: url(#sky-out);"/>
```

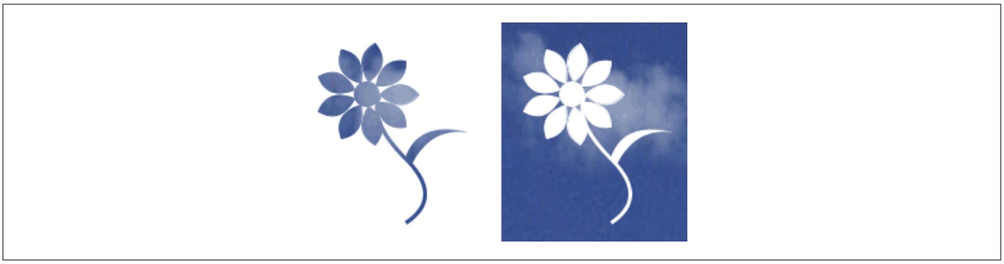


图 11-16：feComposite in 和 out 效果

11.7　<feBlend>滤镜

别急，还有很多！是的，滤镜还提供了另外一种合并图像的方式。`<feBlend>` 元素需要两个输入源，分别指定在 `in` 和 `in2` 属性中，还需要一个 `mode` 属性用于设置如何混合输入源。可能的值有：`normal`、`multiply`、`screen`、`lighten` 和 `darken`。给定一个不透明的输入源：`<feBlend in="A" in2="B" mode="m"/>`，下面描述了每种模式的结果像素的颜色。

- `normal`

  只有 B；和 `<feComposite>` 中的 `over` 运算符一样。

- `multiply`

  顾名思义，对于每个颜色通道，将 A 的值和 B 的值相乘。由于颜色值在 0~1 之间，所以相乘会让它们更小。这会加深颜色，对于暗色或者非常强烈的颜色，效果最强烈；如果某个颜色是白色则没有效果。其效果类似于为两种图像创建一个幻灯片，然后在同一幻灯机中将它们叠放在一起——最后只有同时穿过两者的光线才可见。

- `screen`

  把每个通道的颜色值加在一起，然后减去它们的乘积。明亮的颜色或者浅色往往会比暗色占优势，但是相似亮度的颜色会被合并。这个效果类似于有两台不同的幻灯机，每个图像用一个幻灯机，然后照在同一个屏幕上——一台幻灯机的强者光会压过另一台幻灯机的投影。

- `darken`

  提取 A 和 B 的每个通道的最小值。颜色较暗，因此得名。

- `lighten`

  提取 A 和 B 的每个通道的最大值。颜色较亮，因此得名。

注意，每个红、绿、蓝值的计算都是独立完成的。因此，如果对 RGB 值为 (100%, 0%, 0%) 的纯红色方块和 RGB 值为 (50%, 50%, 50%) 的灰色方块使用 `darken`，结果颜色将会是 (50%, 0%, 0%)。如果输入源不是不透明的，那么所有的模式（除了 `screen` 因素）在计算时都会计算透明度。

最后，一旦颜色值计算完成，结果的透明度就由公式 `1 - (1 - opacity of A) * (1 - opacity of B)` 决定。使用这个公式时，两个不透明项仍然保持不透明，而两个不透明度为 50% 的图形会被合并为一个，不透明度变为 75%。

图 11-17 展示了将一个不透明的从白色到黑色的渐变条分别与不透明的和不透明度为 50% 的颜色块相混合的结果，颜色块的 RGB 值为黑（#000）、黄（#ff0）、红（#f00）、亮绿色（#0c0）以及深蓝色（#009）。

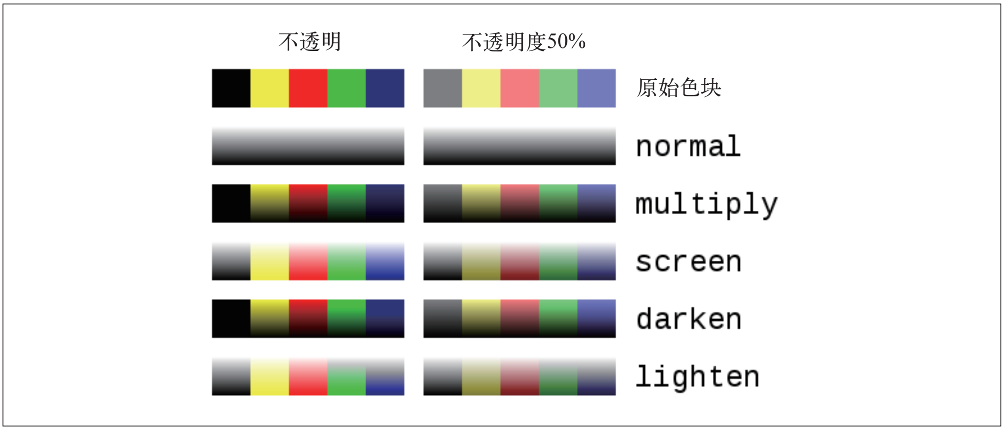


图 11-17：<feBlend> 的效果

11.8 <feFlood>和<feTile>滤镜

<feFlood> 和 <feTile> 元素是很实用的滤镜。它们很像 <feOffset>，允许我们在一系列滤镜基元内执行某些常见的操作，而不是在主图形中创建额外的 SVG 元素。

<feFlood> 提供了一个纯色区域用于组合或者合并。我们只需提供 flood-color 和 flood-opacity，然后滤镜会完成其他工作。

<feTile> 会提取输入信息作为图案，然后横向和纵向平铺填充滤镜指定的区域。图案的尺寸由输入给 <feTile> 的尺寸决定。

示例 11-8 用 <feComposite> 裁剪并平铺了整个花朵的形状区域。用于平铺的图案显示在图 11-18 的右上角用作参考。

示例 11-8：<feFlood> 和 <feTile> 示例

```
<defs>
<filter id="flood-filter" x="0" y="0" width="100%" height="100%">
  <feFlood flood-color="#993300" flood-opacity="0.8" result="tint"/>
  <feComposite in="tint" in2="SourceGraphic"
    operator="in"/>
</filter>

<filter id="tile-filter" x="0" y="0" width="100%" height="100%">
  <feImage xlink:href="cloth.jpg" width="32" height="32"
    result="cloth"/>
  <feTile in="cloth" result="cloth"/>
  <feComposite in="cloth" in2="SourceGraphic"
    operator="in"/>
</filter>

<g id="flower">
```

```
    <!-- 这里绘制花朵图形 -->
</g>
</defs>

<use xlink:href="#flower" transform="translate(0,0)"
  style="filter: url(#flood-filter);"/>
<use xlink:href="#flower" transform="translate(110,0)"
  style="filter: url(#tile-filter);"/>
<image xlink:href="cloth.jpg" x="220" y="10"
  width="32" height="32"/>
```



图 11-18：<feFlood> 和 <feTile> 元素效果

11.9　光照效果

如果用 SVG 绘制一个亮绿色的圆，看起来就像一个从交通信号灯中掉下来的灯，它自己发着光，平躺在屏幕上。而一个从绿纸上裁剪出来的圆形则看起来会更“真实”，因为它的颜色来自外部光照，并且还带有一些纹理。而一个从绿色塑料上裁剪下来的圆不仅有来自外部光照产生的颜色，它本身还反射了一些亮点。外部光照使物体产生颜色称为漫反射；从某个面反射亮点的效果被称作镜面反射。

要得到这些效果，我们必须指定下列信息：

- 想要的反射类型（<feDiffuseLighting> 漫反射或者 <feSpecularLighting> 镜面反射）
- 想要照亮的对象
- 使用的灯光颜色
- 想要的光源类型（<fePointLight> 点光源、<feDistantLight> 远光或者 <feSpotLight> 聚光灯）以及它的位置

要从三个维度指定光源位置。这意味着除了 x 和 y 值之外还需要一个 z 值。正如二维图形一样，x 轴从左向右递增，y 轴从上往下递增。而 z 轴则是朝“离开屏幕”指向我们的方向递增。

这两种光照效果都使用它们照亮对象的阿尔法通道作为凹凸贴图（bump map）；较高的阿尔法值被假定为“凸”在对象表面之上。

11.9.1 漫反射照明

演示 <feDiffuseLighting> 元素工作原理的最好方式就是直接跳到示例 11-9，它在一个绿色的圆形上发出了淡黄色的光，曲线图案纹理来自示例 8-1。

示例 11-9：带有一个点光源的漫反射照明

```
<defs>
  <path id="curve" d="M 0 0 Q 5 20 10 10 T 20 20"     ❶
      style="stroke: black; fill: none;"/>

  <filter id="diff-light" color-interpolation-filters="sRGB"
    x="0" y="0" width="100%" height="100%">    ❷

    <feImage xlink:href="#curve" result="tile"
        width="20" height="20"/>     ❸

    <feTile in="tile" result="tile"/>

    <feDiffuseLighting in="tile"    ❹
        lighting-color="#ffffcc"
        surfaceScale="1"      ❺
        diffuseConstant="0.5"     ❻
        result="diffuseOutput">    ❼
        <fePointLight x="0" y="50" z="50"/>  ❽
    </feDiffuseLighting>     ❾

    <feComposite in="diffuseOutput" in2="SourceGraphic"
        operator="in" result="diffuseOutput"/>    ❿

    <feBlend in="diffuseOutput" in2="SourceGraphic"
        mode="screen"/>    ⓫
  </filter>
</defs>

<circle id="green-light" cx="50" cy="50" r="50"
    style="fill: #060; filter: url(#diff-light)"/>   ⓬
```

❶ 定义用作图案的曲线。

❷ 设置颜色插值方法和滤镜边界。

❸ 用 curve 图像平铺滤镜区域。这会变成凹凸贴图。

❹ 这个平铺区域会输入到 <feDiffuseLighting> 元素中，还给它加了一个浅黄色的光照，正如 lighting-color 属性中所指定的。

❺ surfaceScale 属性代表阿尔法值为 1 时表面的高度（更通用的说法是，它是计算阿尔法值时的乘积因子）。

❻ diffuseConstant 是一个用于确定像素最终 RGB 值的乘积因子。它的值必须大于或等于 0，默认值为 1。要让 lighting-color 更明亮，这个值就应该更小（除非我们想要图片褪色）。

❼ 这个滤镜的结果将被命名为 diffuseOutput。

❽ 这个例子用了一个点光源，意味着光源的光会辐射到所有方向。而我们希望它照亮在滤镜区域的左上角，并且在屏幕的 50 单位前。设置光源离对象越远，对象被照亮得越均匀。在这个例子中，光线距离越近，获得的效果越明显。

❾ `<feDiffuseLighting>` 元素结束。

❿ 用 `<feComposite>` 的 `in` 属性裁剪滤镜的输出到源图形（圆）的边界。

⓫ 最后，`<feBlend>` 设置为 `screen` 模式，它会尝试让源图形变亮，这是滤镜的最后部分。

⓬ 在想要的对象上启用滤镜，生成图 11-19。

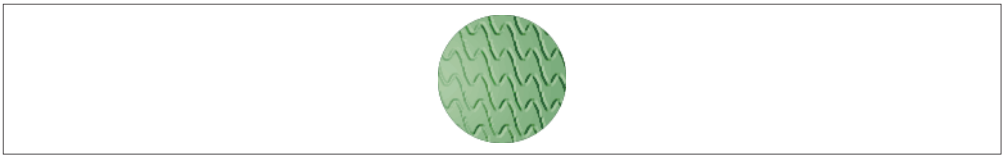


图 11-19：应用漫反射照明的滤镜效果

> 这个滤镜的输入信息是一个包含四个颜色分量的图形，但只有阿尔法通道被使用了。然而，当我（David）插入一个 `<feColorMatrix type="luminanceToAlpha">` 并用它的输出作为滤镜的输入时，并没有得到想要的效果。记住，`luminanceToAlpha` 会把黑色区域（零亮度）转换为完全透明的（零阿尔法）。从阿尔法级别看黑色曲线图案和透明空背景之间毫无差异，所以这个照明效果没有任何对应的纹理。

11.9.2 镜面反射照明

换句话说，镜面照明就是提供亮点而不是照明。示例 11-10 演示了它的工作原理。

示例 11-10：远光镜面反射照明

```
<defs>
  <path id="curve" d="M 0 0 Q 5 20 10 10 T 20 20"
  style="stroke: black; fill: none;"/>      ❶

  <filter id="spec-light" color-interpolation-filters="sRGB"
    x="0" y="0" width="100%" height="100%">      ❷

    <feImage xlink:href="#curve" result="tile"
        width="20" height="20"/>      ❸

    <feTile in="tile" result="tile"/>

    <feSpecularLighting in="tile"      ❹
        lighting-color="#ffffcc"
        surfaceScale="1"      ❺
        specularConstant="1"      ❻
        specularExponent="4"      ❼
```

```
            result="specularOutput">     ❽
            <feDistantLight elevation="25" azimuth="0"/>    ❾
        </feSpecularLighting>    ❿

        <feComposite in="specularOutput" in2="SourceGraphic"
            operator="in" result="specularOutput"/>    ⓫

        <feComposite in="specularOutput" in2="SourceGraphic"
            operator="arithmetic" k1="0" k2="1" k3="1" k4="0"/>    ⓬
      </filter>
    </defs>

    <circle id="green-light" cx="50" cy="50" r="50"
      style="fill: #060; filter: url(#spec-light)"/>    ⓭
```

❶ 和前面的例子一样，定义曲线。

❷ 与前一个例子的唯一区别是滤镜名称。

❸ 和前面的例子一样，这部分平铺曲线。

❹ 开始定义 <feSpecularLighting> 滤镜，指定 lighting-color 为淡黄色。

❺ surfaceScale 属性代表阿尔法值为 1 时表面的高度（更通用的说法是，它是计算阿尔法值时的乘积因子）。

❻ specularConstant 是一个用于确定像素最终 RGB 值的乘积因子。它的值必须大于或等于 0，默认值为 1。要让 lighting-color 更明亮，这个值就应该更小。这个数字的效果也可以通过 specularExponent 属性来缓和。

❼ specularExponent 是用来确定像素最终 RGB 值的另一个因子。这个属性的值必须是 0 到 128 之间的数字，默认值是 1。数字越大，结果越“明亮”。

❽ 这个滤镜的结果将被命名为 specularOutput。

❾ 这个例子用了一个远光源，离图像足够远，因而光照射到图片所有部分的角度都一样。所以这里不是指定光源的位置，而是指定光线来源的角度。

elevation 和 azimuth 属性允许我们从三个维度指定角度。elevation 提供了屏幕平面上光的角度：elevation="0" 表示光线平行于整个图像，而 elevation="90" 表示光线直射下来。

azimuth 属性在平面内指定了角度，当 elevation 为 0 时，azimuth="0" 指定光线从图像右侧来（更普遍的说法是 x 轴正值结束处）；azimuth="90" 表示从底部（y 轴正值结束处）来，azimuth="180" 表示从左侧来，azimuth="270" 表示从顶部来。

❿ <feSpecularLighting> 元素结束。注意，这个滤镜的输入信息是一个阿尔法通道，而输出包含阿尔法和颜色信息（不像 <feDiffuseLighting>，总是生成一个不透明的结果）。

⓫ <feComposite> 的 in 属性裁剪滤镜的输出到源图形（圆）的边界。

⓬ 最后，用 <feComposite> 的 arithmetic 运算符将光照和源图像叠加。

⓭ 在圆形上启动滤镜，生成如图 11-20 所示的高亮浮雕效果。

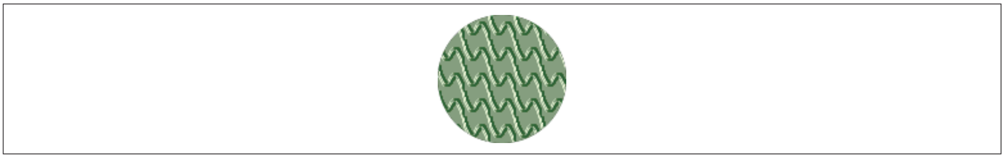


图 11-20：应用镜面照明的滤镜效果

> 关于从三个维度创建光照效果，有一个优秀的教程：http://www.webreference.com/3d/lesson12/。我们只学习了两个维度，但大部分信息都是适用的。

第三种光源类型是 `<feSpotLight>`，用这些属性来指定：`x`、`y` 和 `z`，聚光灯的位置（默认值为 0）；`pointsAtX`、`pointsAtY` 和 `pointsAtZ`，聚光灯指向的地点（默认值为 0）；`specularExponent`，控制光源焦点的值（默认值为 1）；以及 `limitingConeAngle`，用于约束光线投射的范围。这是聚光灯轴和锥形之间的角度。因此，如果我们想要蔓延整个锥 30 度角，指定角度为 15 即可（默认值允许无限蔓延）。

11.10　访问背景

除了 `SourceGraphic` 和 `SourceAlpha` 滤镜输入之外，当我们调用一个滤镜时，滤镜对象还可以访问已经渲染到画布上的图片的某一部分。这部分被称为 `BackgroundImage`（不是 `BackgroundGraphic`）和 `BackgroundAlpha`。为了访问这些输入信息，滤镜对象必须位于 `enable-background` 属性值为 `new` 的容器元素之内。示例 11-11 演示了在背景图像上执行高斯模糊。

示例 11-11：访问背景图像

```
<defs>
  <filter id="blur-background">    ❶
    <feGaussianBlur in="BackgroundImage" stdDeviation="2" result="blur"/>
    <feComposite in="blur" in2="SourceGraphic" operator="in" />
    <feOffset dx="4" dy="4" result="offsetBlur"/>
  </filter>
</defs>

<g enable-background="new">     ❷
  <rect x="0" y="0" width="60" height="60"
    style="fill: lightblue; stroke: blue; stroke-width: 10" />     ❸
  <circle cx="40" cy="40" r="30"
    style="fill: #fff; filter: url(#blur-background);" />    ❹
</g>
```

❶ 类似用于投影的模糊滤镜，只是输入信息是 `BackgroundImage` 而不是 `SourceAlpha`。

❷ 由于 <g> 是一个容器元素，在这里放置 enable-background 是一个完美的选择。它所有的子元素都可以利用背景图像和阿尔法信息。

❸ 把矩形绘制到画布上，然后进入背景缓冲区里。

❹ 圆形不会直接显示；滤镜会模糊背景图像（不包括圆形），然后合成 in 属性值 SourceGraphic。图 11-21 展示了结果。

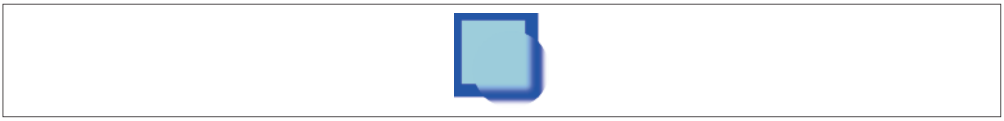


图 11-21：访问背景图像的效果



在写作本书时，还没有浏览器实现 enable-background 或者 BackgroundImage 和 BackgroundAlpha 输入。如果在不支持它们的浏览器中在滤镜内使用这些输入，滤镜不会返回任何信息——这意味着图形的滤镜部分会消失。

一种替代的方式是分离背景到 <g> 元素中，然后用 <feImage> 把它导入到滤镜中。以这种方式修改后的示例 11-11 代码如下：

```
<defs>
  <filter id="blur-background">
    <feImage xlink:href="#background" result="bg"/>
    <feGaussianBlur in="bg" stdDeviation="2" result="blur" />
    <feComposite in="blur" in2="SourceGraphic" operator="in" />
    <feOffset dx="4" dy="4" result="offsetBlur"/>
  </filter>
</defs>

<g id="background">
  <rect x="0" y="0" width="60" height="60"
    style="fill: lightblue; stroke: blue; stroke-width: 10"/>
</g>
<circle cx="40" cy="40" r="30"
    style="fill: #fff; filter: url(#blur-background);" />
```

在支持使用 SVG 片段 <feImage> 的浏览器中，其结果和图 11-21 一样（目前在 Mozilla FireFox 中行不通）。

11.11　<feMorphology>元素

<feMorphology> 元素允许我们对图形进行“瘦身”或者“加厚”。可以指定 operator 值为 erode 来给图形瘦身，或者指定为 dilate 来加厚图形。radius 属性用来告诉我们增厚或者变薄了多少。正如在图 11-22 中可以看到的，在细线条图形上使用瘦身效果会对绘图造成严重的破坏。

示例 11-12：使用 <feMorphology> 对图形进行瘦身或者加厚

http://oreillymedia.github.io/svg-essentials-examples/ch11/fe_morphology.html

```
<defs>
  <g id="cat" stroke-width="2">
     <!-- 这里绘制猫图形 -->
  </g>

  <filter id="erode1">
    <feMorphology operator="erode" radius="1"/>
  </filter>

  <filter id="dilate2">
    <feMorphology operator="dilate" radius="2"/>
  </filter>
</defs>

<use xlink:href="#cat"/>
<text x="75" y="170" style="text-anchor: middle;">Normal</text>

<use xlink:href="#cat" transform="translate(150,0)"
    style="filter: url(#erode1);"/>
<text x="225" y="170" style="text-anchor: middle;">Erode 1</text>

<use xlink:href="#cat" transform="translate(300,0)"
    style="filter: url(#dilate2);"/>
<text x="375" y="170" style="text-anchor: middle;">Dilate 2</text>
```

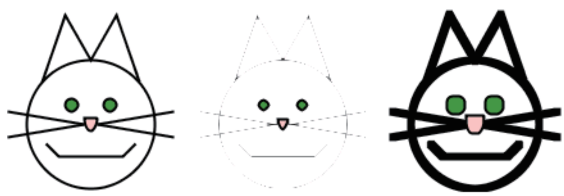


图 11-22：使用 <feMorphology> 的效果

11.12 <feConvolveMatrix>元素

<feConvolveMatrix> 元素允许我们按照它邻近的像素计算像素的新值。这个滤镜允许我们生成诸如模糊、锐化、浮雕和斜切这样的效果。它的原理是合并像素和它邻近的像素，生成结果像素值。想象一下像素 P 和其 8 个邻近的像素如下所示（这个滤镜的常见情况）：

```
A B C
D P E
F G H
```

在 kernelMatrix 属性中指定 9 个因数即可。这些数字代表每个像素乘以多少。这些结果会

被累加，总数很有可能大于 1（比如当所有的因数都是正值时），因此，为了均匀强度，结果还要除以因数总和。假设我们指定如下 9 个数字（为了展示它们是一个矩阵，数字之进行了间隔）：

```
<feConvolveMatrix kernelMatrix="
  0  1  2
  3  4  5
  6  7  8"/>
```

像素 P 的新值将是如下所示：

```
P' = ((0*A) + (1*B) + (2*C) +
     (3*D) + (4*P) + (5*E) +
     (6*F) + (7*G) + (8*H)) / (0 + 1 + 2 + 3 + 4 + 5 + 6 + 7 + 8)
```

唯一的例外是如果所有矩阵值总和为 0，则不会执行除法。

我们还可以指定一个 bias 属性，它通过为每个像素添加指定的偏移值来改变滤镜的输出范围。bias 会在除法后面计算，但是在最终调整结果到 0~1 这个有效范围之前。

示例 11-13 通过每个像素的左上角减去右下角实现了如图 11-23 所示的浮雕效果。[3]0.5 的 bias 值会在 kernalMatrix 进行求和和除法运算之后加上去。偏差值导致那些左上角和右下角邻近像素相同的像素显示为灰色（与没有偏差值改变的黑色不同）。显示暗色时，左上角的邻近像素比左下角的邻近像素更亮，而显示亮色时，左下角的邻近像素更亮。结果，对角边缘被突出显示，就像图像被举起并从旁边照亮了。透明的背景像素被认为是黑色。

<feConvolveMatrix> 的默认行为是对所有颜色分量进行计算，包括阿尔法通道。在这个前提下，只有每个形状的边缘部分，即左上角的邻近像素的阿尔法值比右下角邻近像素的阿尔法值更高时，才会显示出来。为了只对红、绿、蓝三个值进行计算，要指定 preserveAlpha 为 true；它的默认值为 false。

示例 11-13：<feConvolveMatrix> 的浮雕效果

```
<defs>
  <filter id="emboss">
    <feConvolveMatrix
      preserveAlpha="true"
      kernelMatrix="1 0 0 0 0 0 0 0 -1"
      bias="0.5"/>
  </filter>

  <g id="flower">
    <!-- 这里绘制花朵图形 -->
  </g>
```

注 3：在 1.7 版的 Apache Batik 上测试时，滤镜包含元素会导致渲染错误；在浏览器中进行测试时这个例子会按照预期运行。

```
</defs>

<use xlink:href="#flower" style="filter: url(#emboss);"/>
```

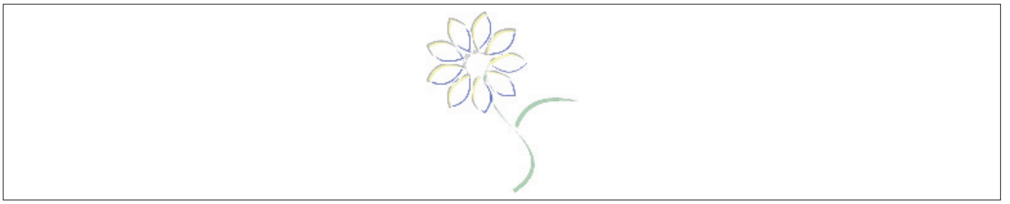


图 11-23：使用 <feConvolveMatrix> 的效果

虽然默认矩阵规格是三列三行，但我们可以用 order 属性指定任意想要的规格。如果指定 order="4"，那么矩阵的 kernelMatrix 属性中就需要 16 个数字（4 乘 4）。三列两行的矩阵要通过 order="3 2" 的形式指定，需要 6 个数字。矩阵越大，生成结果所需要的计算就越多。

对于图形中间的像素，很容易确定邻近像素。那么图形边缘的像素怎么办？它们的邻近像素是什么？这由我们提供的 edgeMode 属性决定。如果设置它的值为 duplicate（默认值），那么 <feConvolveMatrix> 会复制所需方向上的边缘值生成邻近值。值 wrap 则会绕到相反的一侧找到邻近值。比如，顶部像素的邻近值在底部，左边缘像素左侧邻近值的像素在右边缘。如果图片被用作重复的平铺图案，则这一行为很有用。值 none 会为所有缺失的邻近值提供一个透明的黑色像素（红、绿、蓝和阿尔法值都为 0）。

<feConvolveMatrix> 可能产生的所有效果，这里不可能一一描述。可以在在线示例中进行尝试，看看会产生什么效果：http://oreillymedia.github.io/svg-essentials-examples/ch11/convolve.html。

11.13 <feDisplacementMap>元素

这个神奇的滤镜使用第二个输入的颜色值决定在第一个输入中移动像素的距离。我们可以用 xChannelSelector 属性指定应该用哪个颜色通道来影响像素的 *x* 坐标，用 yChannelSelector 属性指定用哪个颜色通道来影响 y 坐标。这些选择器的合法值是 "R"、"G"、"B"、"A"（也就是阿尔法通道）。还必须指定移动像素的距离；scale 属性用于指定缩放因子。如果不指定这个属性，滤镜什么都不做。

示例 11-14 中创建了一个渐变矩形作为第二个输入。位移缩放因子设置为 10，红色通道会被用作 *x* 偏移，绿色通道会被用作 *y* 偏移。图 11-24 展示了给花朵应用这个位移的结果。

示例 11-14：使用渐变作为位移映射

```
<defs>
  <linearGradient id="gradient">
    <stop offset="0" style="stop-color: #ff0000;" />
```

```
      <stop offset="0.5" style="stop-color: #00ff00;"/>
      <stop offset="1" style="stop-color: #000000;"/>
    </linearGradient>

    <rect id="rectangle" x="0" y="0" width="100" height="200"
      style="fill: url(#gradient);"/>

    <filter id="displace">
      <feImage xlink:href="#rectangle" result="grad"/>

      <feDisplacementMap
        scale="10"
        xChannelSelector="R"
        yChannelSelector="G"
        in="SourceGraphic" in2="grad"/>
    </filter>
    <g id="flower">
      <!-- 这里绘制花朵图形 -->
    </g>
  </defs>

  <use xlink:href="#flower" style="filter: url(#displace);"/>
```

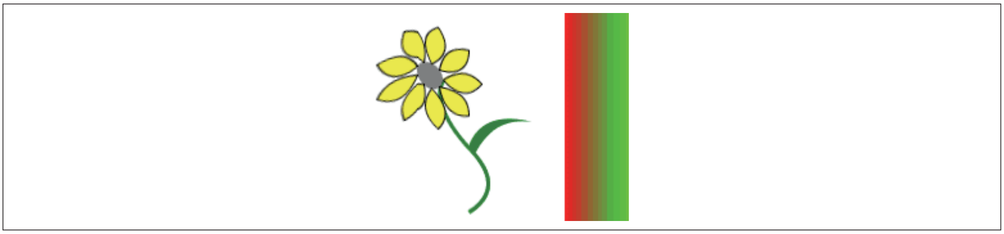


图 11-24：使用 <feDisplacementMap> 的效果

还可以对两个输入使用同一图形。这意味着图形的位移由它自己的着色控制。示例 11-14 的效果如图 11-25 所示，相当古怪。

示例 11-15：使用图形作为自身的位移映射

```
  <defs>
  <filter id="self-displace">
    <feDisplacementMap
      scale="10"
      xChannelSelector="R"
      yChannelSelector="G"
      in="SourceGraphic" in2="SourceGraphic"/>
  </filter>

  <g id="flower">
    <!-- 这里绘制花朵图形 -->
  </g>
  </defs>

  <use xlink:href="#flower" style="filter: url(#self-displace);"/>
```

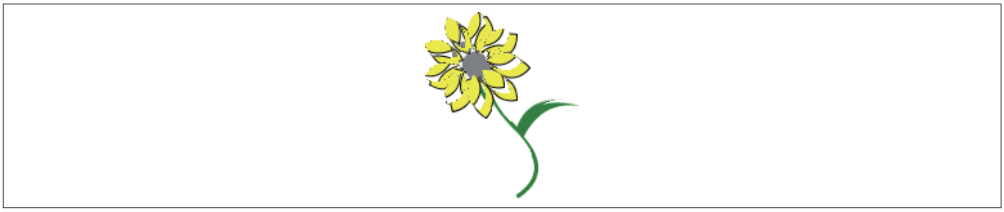


图 11-25：同一图形用作两个输入的效果

11.14 <feTurbulence>元素

`<feTurbulence>` 元素允许我们通过使用由 Ken Perlin 开发的方程，生成大理石、云彩等人工纹理效果。这个方程被称作 Perlin noise（http://freespace.virgin.net/hugo.elias/models/m_perlin.htm）。我们要指定以下属性。

- `type`

  `turbulence` 和 `fractalNoise` 之一。后者显示更平滑。

- `baseFrequency`

  给这个属性值的数字越大，结果颜色的变化越快。这个数值必须大于 0 且应该小于 1。还可以给这个属性提供两个数字，第一个将被作为 *x* 方向的频率，而第二个将被作为 *y* 方向的频率。

- `numOctaves`

  这是噪音函数使用的数值，生成最终结果时应该加上这个数值。数值越大，纹理粒度越细。默认值为 1。

- `seed`

  这个滤镜使用的随机数生成器的种子。默认值为 0；改变它可以得到一些不同的结果。

图 11-26 是一个 SVG 文件演示前三个属性的不同值的截图。

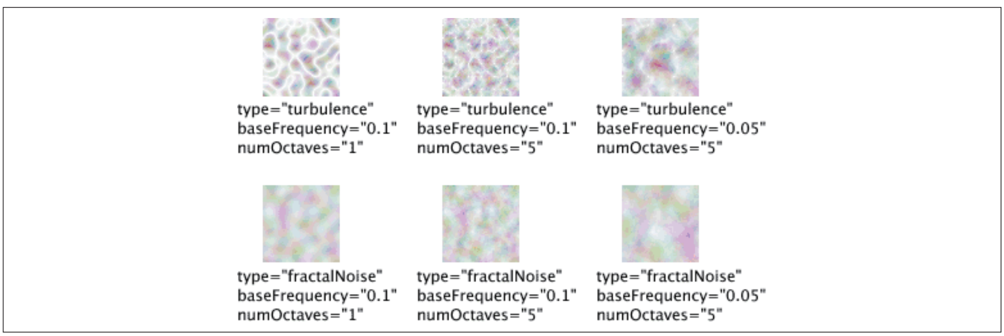


图 11-26：`<feTurbulence>` 属性的各种值

在线示例中可以尝试所有这三个参数：

http://oreillymedia.github.io/svg-essentials-examples/ch11/turbulence.html

11.15 滤镜总结

`<filter>` 元素包含一系列滤镜基元，每个都接受一个或多个输入，同时提供了唯一的结果供其他滤镜使用。一系列滤镜中最后一个滤镜的结果会呈现到最终的图形上。我们用 `x`、`y`、`width` 和 `height` 属性指定应用滤镜的画布的尺寸。用 `filterUnits` 指定用来定义滤镜范围的单位，用 `primitiveUnits` 为滤镜基元中的各种长度值指定坐标系统。

表 11-1 进行了滤镜总结。每个滤镜基元元素都有一个 `in` 属性用来提供基元来源，还可以指定 `x`、`y`、`width` 和 `height`。

表11-1：滤镜总结

元　素	属　性
`<feBlend>`	`in2="second source"` `mode="normal" | "multiply" | "screen" | "darken" | "lighten"`（默认是 `normal`）
`<feColorMatrix>`	`type="matrix" | "saturate" | "hueRotate" | "luminanceToAlpha"` `values="matrix values" | "saturation value(0-1) | "rotate degrees"`
`<feComponentTransfer>`	`<feFuncR>`、`<feFuncG>`、`<feFuncB>` 和 `<feFuncA>` 元素的容器

`<feFunc`*X*`>`	`type="identity" | "table" | "discrete" | "linear" | "gamma"`
	`tableValues="intervals for table, steps for dicrete"`
	`slope="linear slope"`
	`intercept="linear intercept"`
	`amplitude="gamma amplitude"`
	`exponent="gamma exponent"`
	`offset="gamma offset"`

`<feComposite>`	`in2="second source"`
	`operator="over" | "in" | "out" | "atop" | "xor" | "arithmetic"`
	下列属性用于 `arithmetic`（没指定的属性的默认值都为 0）：
	`k1="factor for in1 * in2"`
	`k2="factor for in1"`
	`k3="factor for in2"`
	`k4="additive offset"`

`<feConvolveMatrix>`	`order="colums rows"`（默认为 3 × 3）
	`kernel="values"`
	`bias="offset value"`

`<feDiffuseLighting>`	光源元素的容器 `surfaceScale="height"`（默认为 1） `diffuseConstant="factor"`（必须为正数，默认为 1）

（续）

元　素	属　性
<feDisplacementMap>	scale="displacement factor"（默认为 0） xChannelSelector="R" | "G" | "B" | "A" yChannelSelector="R" | "G" | "B" | "A" in2="second input"
<feFlood>	flood-color="color specification" flood-opacity="value (0-1)"
<feGaussianBlur>	stdDeviation="blur spread"（值越大越模糊，默认为 0）
<feImage>	xlink:href="/getfile?safari4=trueitem=/images/9781491945308/assets/image source"
<feMerge>	<feMergeNode> 元素的容器
<feMergeNode>	in="intermediate result"
<feMorphology>	operator="erode" | "dilate" radius="x-radius y-radius" radius="radius"
<feOffset>	dx="x offset"（默认为 0） dy="y offset"（默认为 0）
<feSpecularLighting>	光源元素的容器 surfaceScale="height"（默认为 1） specularConstant="factor"（必须为正数，默认为 1） specularExponent="exponent"（1~128 之间的值，默认为 1）
<feTile>	in 的平铺图案
<feTurbulence>	type="turbulence" | "fractalNoise" baseFrequency="x-frequency y-frequency" baseFrequency="frequency" numOctaves="integer" seed="number"
<feDistantLight>	azimuth="degrees"（默认为 0） elevation="degrees（默认为 0）
<fePointLight>	x="coordinate"（默认为 0） y="coordinate"（默认为 0） z="coordinate"（默认为 0）
<feSpotLight>	x="coordinate"（默认为 0） y="coordinate"（默认为 0） z="coordinate"（默认为 0） pointsAtX="coordinate"（默认为 0） pointsAtY="coordinate"（默认为 0） pointsAtZ="coordinate"（默认为 0） specularConstant="focus control"（默认为 1） limitingConeAngle="degrees"

第 12 章

SVG动画

到目前为止，我们所看到的图像都是静态图像，一旦构建好就永远不会改变。本章，我们将会分析两种让图像动起来的方法。第一种方法是基于 SMIL 的动画，应该用于描述构成图形基本组成部分的动画，且这个动画可以提前定义好。CSS 动画应该用于一些风格效果和简单的反馈（比如聚焦或者悬停时高亮元素）。脚本应该用于更复杂的交互，我们会在第 13 章中讲述。

上一章，我们建议将滤镜用作增强图形信息的一种手段，而不应作为结果。这一建议对动画来说更为重要。借助动画的能力，你可能会想要把每一个图形都变成百老汇那样载歌载舞的景象。如果你的目标只是实验，这并没有什么问题。但是，如果你的目标是传达一个信息，那么使用不必要的动画或者过度使用动画都是非常糟糕的。我们再讲清楚点儿：除了公司的 CEO 之外，没有人有兴趣重复观看一个旋转的、闪烁的、颜色不断变化的、闪光灯式的公司 Logo。

本章主要讲述动画，因此大部分例子将不使用任何内容，当然，也会尽量避免不必要的或者过分雕琢的动画。

> IE 浏览器（包括 IE11，编写本章时的最新版）还不支持基于 SMIL 的动画或者给 SVG 元素应用动画。但是基于 JavaScript 的解决方案，比如 SMILscript（http://schepers.cc/svg/smilscript/）以及 FakeSMILe（http://leunen.me/fakesmile/），可以把基于 SMIL 的动画转换为基于脚本的动画在 IE 中使用。

12.1 动画基础

SVG 的动画特性基于万维网联盟的“同步多媒体集成语言”（SMIL）规范（http://www.w3.org/TR/SMIL3/）。在这个动画系统中，我们可以指定想要进行动画的属性（比如颜色、动作或者变形属性）的起始值和结束值，以及动画的开始时间和持续时间。示例 12-1 给出了一个基本示例代码。

示例 12-1：收缩矩形

http://oreillymedia.github.io/svg-essentials-examples/ch12/simple_animation.html

```
<rect x="10" y="10" width="200" height="20" stroke="black" fill="none">
  <animate
    attributeName="width"
    attributeType="XML"
    from="200" to="20"
    begin="0s" dur="5s"
    fill="freeze" />
</rect>
```

首先需要注意的是 `<rect>` 元素不再是一个空元素，它里面包含了动画元素。

`<animate>` 元素指定了下列信息。

- `attributeName`，动画中应该持续改变的值；在这里就是 `width`。
- `attributeType`。`width` 属性是一个 XML 属性。另一个常用的 `attributeType` 值是 `CSS`，表示我们想要改变的属性是一个 CSS 属性。如果忽略这一属性，它的默认值是 `auto`；它首先会搜索 CSS 属性，然后才是 XML 属性。
- 属性的起始（`from`）和结束（`to`）值。在这个例子中，起始值是 200，结束值是 20。`from` 值是可选的；如果不指定，则会使用父元素的值。此外，还有一个 `by` 属性，可以代替 `to`，它是一个从 `from` 值开始的偏移量；动画结束时属性的值为结束值。
- 动画的开始时间和持续时间。在这个例子中，时间以秒为单位，通过在数字后面使用 `s` 指定。定义时间的其他方式会在 12.2 节中描述。
- 动画结束时做什么。在这个例子中，持续 5 秒之后，属性会“冻结”（freeze）为 `to` 值。也就是 SMIL `fill` 属性，它会告诉动画引擎如何填补剩下的时间。不要把它跟 SVG 的 `fill` 属性混淆了，该属性用于告诉 SVG 如何描绘对象。如果我们移除这一行，会使用默认值（`remove`），5 秒的动画完成之后 `width` 属性会返回它的原始值 200。

图 12-1 和图 12-2 展示了动画的开始和结束阶段。它们并不能很好地展示实际效果，因此我们强烈建议你在浏览器中试试。

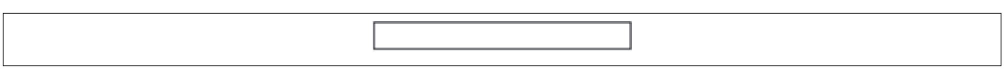


图 12-1：动画开始

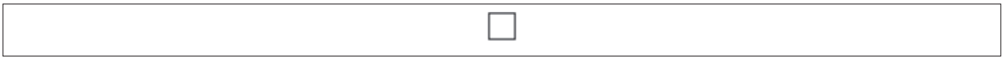


图 12-2：动画结束

示例 12-2 更复杂一些。它从一个 20 乘 20 的绿色方块开始，将在 8 秒的时间里增长为 250 乘 200。前 3 秒，绿色的透明度会增加，接下来 3 秒会减小。注意 `fill-opacity` 是使用 `attributeType="CSS"` 的，因为它设置在 `style` 属性中。

示例 12-2：单个对象上的多重动画

http://oreillymedia.github.io/svg-essentials-examples/ch12/multiple_animation.html

```
<rect x="10" y="10" width="20" height="20"
  style="stroke: black; fill: green; style: fill-opacity: 0.25;">
  <animate attributeName="width" attributeType="XML"
    from="20" to="200" begin="0s" dur="8s" fill="freeze"/>
  <animate attributeName="height" attributeType="XML"
    from="20" to="150" begin="0s" dur="8s" fill="freeze"/>
  <animate attributeName="fill-opacity" attributeType="CSS"
    from="0.25" to="1" begin="0s" dur="3s" fill="freeze"/>
  <animate attributeName="fill-opacity" attributeType="CSS"
    from="1" to="0.25" begin="3s" dur="3s" fill="freeze"/>
</rect>
```

最后一个简单的例子是示例 12-3，一个正方形和圆形动画。正方形尺寸会在 8 秒的时间里从 20 乘 20 扩大到 120 乘 120。动画开始 2 秒之后，圆形的半径在 4 秒的时间里从 20 扩大到 50。图 12-3 展示了 4 个时间里的动画组合截图：0 秒，动画开始时；2 秒，圆形开始扩大时；6 秒，圆形完成扩大时；8 秒，动画结束时。

示例 12-3：多个对象的简单动画

http://oreillymedia.github.io/svg-essentials-examples/ch12/multiple_animation2.html

```
<rect x="10" y="10" width="20" height="20"
  style="stroke: black; fill: #cfc;">
  <animate attributeName="width" attributeType="XML"
     begin="0s" dur="8s" from="20" to="120" fill="freeze"/>
  <animate attributeName="height" attributeType="XML"
     begin="0s" dur="8s" from="20" to="120" fill="freeze"/>
</rect>

<circle cx="70" cy="70" r="20"
  style="fill: #ccf; stroke: black;">
  <animate attributeName="r" attributeType="XML"
    begin="2s" dur="4s" from="20" to="50" fill="freeze"/>
</circle>
```

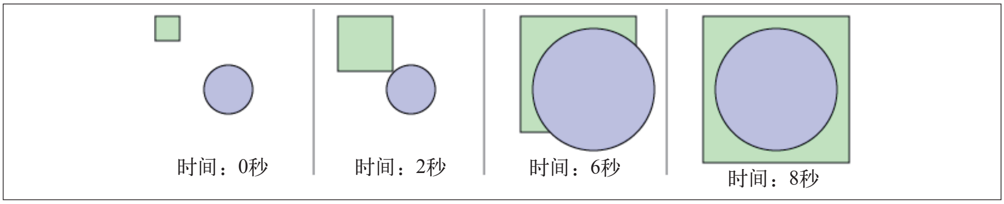


图 12-3：多重对象动画的各阶段

12.2　动画时间详解

SVG 动画用到的动画时钟在 SVG 加载完成时开始启动计时，当用户离开页面时停止计时。我们可以以下列任意一种方式指定特定动画的开始和持续时间为一个数值。

- 时、分、秒形式（`1:20:23`）的完整时间值。
- 分、秒形式（`02:15`）的局部时间值。
- 以 `h`（时）、`min`（分）、`s`（秒）或者 `ms`（毫秒）缩写结尾的时间值，比如 `dur="3.5s" begin="1min"`。不可以在值和单位之间插入任何空白。

如果不指定单位，默认为秒。

12.3　同步动画

我们可以绑定动画的开始时间为另一个动画的开始或者结束，而不是在文档加载时定义每个动画的开始时间。示例 12-4 中有两个圆的动画，第二个会在第一个停止缩放时开始扩大。图 12-4 展示了该动画的重要阶段。

示例 12-4：同步动画

```
<circle cx="60" cy="60" r="30" style="fill: #f9f; stroke: gray;">
  <animate id="c1" attributeName="r" attributeType="XML"
    begin="0s" dur="4s" from="30" to="10" fill="freeze"/>
</circle>

<circle cx="120" cy="60" r="10" style="fill: #9f9; stroke: gray;">
  <animate attributeName="r" attributeType="XML"
    begin="c1.end" dur="4s" from="10" to="30" fill="freeze"/>
</circle>
```

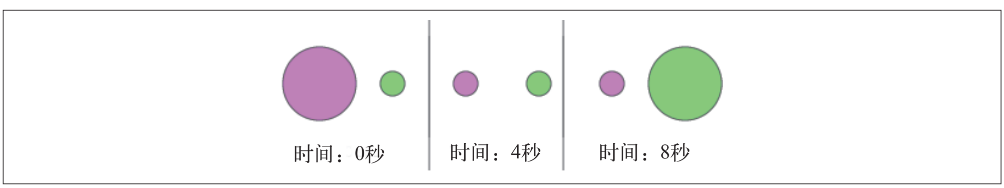


图 12-4：同步动画的各阶段示意图

我们还可以给同步添加一个偏移量。要让一个动画在另一个动画开始 2 秒后再开始，我们要使用类似 begin="otherAnim.end+2s" 的形式指定（可以在加号两侧添加空白）。示例 12-5 中，第二个圆会在第一个圆开始收缩 1.25 秒之后开始增长。

示例 12-5：带有偏移量的同步动画

http://oreillymedia.github.io/svg-essentials-examples/ch12/sync_with_offset.html

```
<circle cx="60" cy="60" r="30" style="fill: #f9f; stroke: gray;">
  <animate id="c1" attributeName="r" attributeType="XML"
    begin="0s" dur="4s" from="30" to="10" fill="freeze"/>
</circle>

<circle cx="120" cy="60" r="10" style="fill: #9f9; stroke: gray;">
  <animate attributeName="r" attributeType="XML"
    begin="c1.begin+1.25s" dur="4s" from="10" to="30" fill="freeze"/>
</circle>
```

在介绍完同步动画之后，我们终于可以介绍 end 属性了。它可以给动画设置一个结束时间，但这并不能替代 dur 属性！下面这个动画会在页面载入 6 秒之后开始。它会持续 12 秒或者直到名为 otherAnim 的动画结束，这取决于是动画先结束还是 12 秒先到：

```
<animate attributeName="width" attributeType="XML"
  begin="6s" dur="12s" end="otherAnim.end"
  from="10" to="100" fill="freeze"/>
```

当然，我们也可以设置 end 的值为一个指定的时间；这可以用于在中途拦截动画，这样我们就可以看看是否一切都在正确的位置。这也是我们创建图 12-3 的方式。下面这个动画在 5 秒时开始，并且应该持续 10 秒，但是会在文档载入后 9 秒（动画开始 4 秒后）暂停。这个动画会停止在 40% 处，因此宽度会冻结在 140（从 100 到 200 的 40% 的距离）。

```
<animate attributeName="width" attributeType="XML"
    begin="5s" dur="10s" end="9s"
    from="100" to="200" fill="freeze"/>
```

12.4 重复动作

到目前为止，动画都只发生一次；fill 被设置为 freeze 使动画保持在最后阶段。如果想要对象返回到动画起始状态，忽略这个属性即可（等价于设置 fill 为默认值 remove）。

另外两个属性允许我们重复动画。第一个是 repeatCount，设置一个整型值，告诉引擎我们想要将指定的动画重复多少次。第二个是 repeatDur，设置一个值，告诉引擎重复应该持续多长时间。如果想要动画重复到用户离开页面，设置 repeatCount 或者 repeatDur 的值为 indefinite 即可。通常我们只会使用其中一个，而不是两个都使用。如果同时指定 repeatCount 和 repeatDur 两个属性，则哪个属性指定的时间先到达就使用哪个属性。

示例 12-6 中的动画展示了两个圆形。上面的圆从左侧移到右侧，重复 2 次，每次 5 秒。第二个圆从右侧移动到左侧，总共持续 8 秒。

示例 12-6：重复动画示例

http://oreillymedia.github.io/svg-essentials-examples/ch12/repeated_action.html

```
<circle cx="60" cy="60" r="30" style="fill: none; stroke: red;">
  <animate attributeName="cx" attributeType="XML"
    begin="0s" dur="5s" repeatCount="2"
    from="60" to="260" fill="freeze"/>
</circle>

<circle cx="260" cy="90" r="30" style="fill: #ccf; stroke: black;">
  <animate attributeName="cx" attributeType="XML"
    begin="0s" dur="5s" repeatDur="8s"
    from="260" to="60" fill="freeze"/>
</circle>
```

就像我们可以相对于另一个动画的开始或者结束时间来同步动画一样，我们也可以将某个动画的开始时间绑定到另一个重复动画的第指定次数动画开始时间。要这样做的话，需要给第一个动画一个 `id`，然后设置第二个动画的 `begin` 为 `id.repeat(count)`，`count` 是第一个动画的重复次数，以 0 开始。示例 12-7 中展示了上面的圆从左侧移到右侧三次，每次重复需要 5 秒。下面的正方形会从右侧移到左侧一次，但是直到上面的圆重复第二次之后 2.5 秒时才开始。

示例 12-7：带重复的同步动画

http://oreillymedia.github.io/svg-essentials-examples/ch12/sync_repetition.html

```
<circle cx="60" cy="60" r="15"
  style="fill: none; stroke: red;">
  <animate id="circleAnim" attributeName="cx" attributeType="XML"
    begin="0s" dur="5s" repeatCount="3"
    from="60" to="260" fill="freeze"/>
</circle>

<rect x="230" y="80" width="30" height="30"
  style="fill: #ccf; stroke: black;">
  <animate attributeName="x" attributeType="XML"
    begin="circleAnim.repeat(1)+2.5s" dur="5s"
    from="230" to="30" fill="freeze"/>
</rect>
```

12.5　对复杂的属性应用动画

动画并不仅限于简单的数值和长度。我们可以为几乎任何属性和样式应用动画，使这些属性在两个值之间平滑过渡。

要为颜色应用动画，只需要给 `from` 和 `to` 属性应用有效的颜色即可，颜色值如 4.2.2 节中

所描述。颜色被看作是用于计算的三个数值分量：R、G、B 的每个值都会从一个颜色转换到另一个。[1] 示例 12-8 为圆形的填充和笔画颜色应用了动画，填充色从淡黄色改变为红色，轮廓从灰色改变为蓝色。这两个动画都是在页面载入 2 秒之后开始，这为我们查看原始颜色提供了时间。

示例 12-8：颜色动画示例

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_color.html

```
<circle cx="60" cy="60" r="30"
        style="fill: #ff9; stroke: gray; stroke-width: 10;">
  <animate attributeName="fill"
    begin="2s" dur="4s" from="#ff9" to="red" fill="freeze"/>
  <animate attributeName="stroke"
    begin="2s" dur="4s" from="gray" to="blue" fill="freeze"/>
</circle>
```

我们还可以为值为数字列表的属性应用动画，只要列表中数字的数量没有改变即可；列表中的每个值都是单独变换的。这意味着我们可以为路径数据或者多边形的点应用动画，只要我们维持点的数量和路径片段的类型即可；示例 12-9 展示了一个 <polygon> 和一个 <path> 动画。

示例 12-9：路径和多边形动画示例

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_path_poly.html

```
<polygon points="30 30 70 30 90 70 10 70"
  style="fill:#fcc; stroke:black">
  <animate id="animation"
    attributeName="points"
    attributeType="XML"
    to="50 30 70 50 50 90 30 50"
    begin="0s" dur="5s" fill="freeze" />
</polygon>

<path d="M15 50 Q 40 15, 50 50, 65 32, 100 40"
  style="fill:none; stroke: black" transform="translate(0,50)">
  <animate attributeName="d"
    attributeType="XML"
    to="M50 15 Q 15 40, 50 50, 32 65, 40 100"
    begin="0s" dur="5s" fill="freeze"/>
</path>
```

12.6 指定多个值

目前为止呈现的所有动画元素都提供了一个起始（from 或默认）值和结束（to）值，然后

注 1：正如 11.5 节中“定义色彩空间”部分所表述的，颜色变化受 color-interpolation 属性的影响。默认使用 sRGB 计算插值，通常会生成合适的结果。

让计算机计算如何从起始值到结束值。我们还可以给动画提供一个特定的中间值，从而允许用一个独立的 <animate> 元素定义复杂的变化序列。我们可以提供一个用分号分隔的、动画在持续时间内使用的值列表，而不是如示例 12-8 中那样按照从淡黄色到红色的方式绘制颜色。示例 12-10 展示了一个使用淡黄色、淡蓝色、粉红色以及淡绿色值动画绘制颜色的圆。

示例 12-20：使用特定的值动画绘制颜色

http://oreillymedia.github.io/svg-essentials-examples/ch12/animating_values.html

```
<circle cx="50" cy="50" r="30"
  style="fill: #ff9; stroke:black;">
  <animate attributeName="fill"
    begin="2s" dur="4s" values="#ff9;#99f;#f99;#9f9"
    fill="freeze"/>
</circle>
```

`values` 属性也可以用来实现交替来回的重复动画，即从起始值到结束值再回到起始值，使用 `values="start; end; start;"` 形式即可。

12.7　多级动画时间

当一个动画有多个值时，动画的持续时间（`dur` 属性）就是要依次通过所有值的时间。默认情况下，动画的持续时间被划分为每个过渡周期等长的片段。示例 12-10 中使用了四个值，因此有三个颜色过渡；总的持续时间为 4 秒，因此每个过渡持续 4/3 秒。

`keyTimes` 属性允许我们以其他方式划分持续时间。`keyTimes` 的格式也是一个分号分隔的列表，但它必须有和 `values` 相同数目的条目。第一个条目始终为 0，最后一个始终为 1；中间时间使用 0 到 1 之间的小数表示，代表动画的持续时间比例，当动画到达某个值的时候刚好对应相应的时间值。

对时间的更多控制可以通过 `calcMode` 属性完成。`calcMode` 有以下四个可能的值。

- `paced`

  SVG 阅读器会计算后续各个值之间的间隔，并以此为依据划分持续时间，因此其变化的速度是恒定的（`keyTimes` 属性会被忽略）。适用于颜色、简单的数值或者长度，但不能用于点列表或者路径数据。

- `linear`

  `<animate>` 元素的默认行为，每个过渡内的速度是恒定的，但是分配给每个过渡的时长是相等的（如果没指定 `keyTimes`）或者由 `keyTimes` 决定。

- `discrete`

  动画会从一个值跳到另一个值，但没有过渡。如果动画绘制一个不支持过渡的属性（比如 `font-family`），会自动使用该模式。

- `spline`

  动画会按照 `keySplines` 属性加速或者减速；更多信息可以阅读 SVG 规范（http://www.w3.org/TR/SVG11/animate.html#KeySplinesAttribute）。

12.8 <set>元素

所有这些动画归根结底都是修改值。有时候，特别是对于非数字属性或者不能过渡的属性，我们可能想要在动画序列的某个点上改变某个值。

比如，我们可能想要一个初始不可见的文本项，使它在某个时间变得可见；这里并不需要 `from` 和 `to`。因此，SVG 引入了方便速记的 `<set>` 元素，它只需要一个 `to` 属性以及适当的时间信息。示例 12-11 将一个圆缩小为 0，然后在圆消失半秒之后显示出了文本。

示例 12-11：<set> 元素示例

http://oreillymedia.github.io/svg-essentials-examples/ch12/animation_set.html

```
<circle cx="60" cy="60" r="30" style="fill: #ff9; stroke: gray;">
  <animate id="c1" attributeName="r" attributeType="XML"
    begin="0s" dur="4s" from="30" to="0" fill="freeze"/>
</circle>

<text text-anchor="middle" x="60" y="60" style="visibility: hidden;">
  <set attributeName="visibility" attributeType="CSS"
    to="visible" begin="4.5s" dur="1s" fill="freeze"/>
  All gone!
</text>
```

12.9 <animateTransform>元素

`<animate>` 元素并不适用于旋转、平移、缩放或倾斜变换，因为它们都“被包裹”在 `transform` 属性内。`<animateTransform>` 元素就是用来解决这个问题的。我们可以设置它的 `attributeName` 为 `transform`。然后用 `type` 属性的值指定变换的哪个值应该变化（`translate`、`scale`、`rotate`、`skewX` 或者 `skewY` 之一）。`from` 和 `to` 的值指定为适当的要动画绘制的变换。撰写本文时，大多数实现当前都只支持在 XML `transform` 属性上使用 `<animateTransform>` 而不支持 CSS3 变换。

示例 12-12 把矩形在水平方向上由正常的比例拉伸了四倍，垂直方向上拉伸了两倍。注意矩形是围绕原点的，因此它在缩放时并不会移动；而它在一个 `<g>` 元素内，因此它可以被

平移到更适当的位置。图 12-5 展示了动画的开始和结束。

示例 12-12：<animateTransform> 示例

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_transform.html

```
<g transform="translate(100,60)">
  <rect x="-10" y="-10" width="20" height="20"
    style="fill: #ff9; stroke: black;">
    <animateTransform attributeType="XML"
      attributeName="transform" type="scale"
      from="1" to="4 2"
      begin="0s" dur="4s" fill="freeze"/>
  </rect>
</g>
```

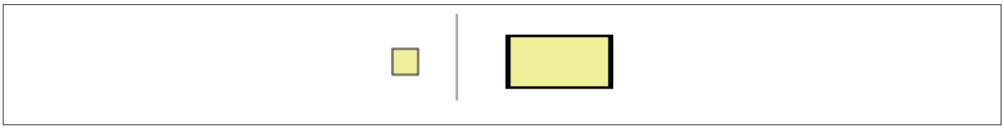


图 12-5：<animateTransform> 之前和之后

如果打算为多个变换应用动画，必须使用 additive 属性。additive 的默认值为 replace，它会替换动画对象的指定变换。但这不适用于一系列的变换，因为后面的动画设置的变换会覆盖前面的。通过设置 additive 为 sum，SVG 会积累变换。示例 12-13 拉伸并旋转了矩形。变换前后的图形如图 12-6 所示。

示例 12-13：多个 <animateTransform> 元素示例

http://oreillymedia.github.io/svg-essentials-examples/ch12/additive_transform.html

```
<rect x="-10" y="-10" width="20" height="20"
  style="fill: #ff9; stroke: black;">
  <animateTransform attributeName="transform" attributeType="XML"
    type="scale" from="1" to="4 2"
    additive="sum" begin="0s" dur="4s" fill="freeze"/>
  <animateTransform attributeName="transform" attributeType="XML"
    type="rotate" from="0" to="45"
    additive="sum" begin="0s" dur="4s" fill="freeze"/>
</rect>
```

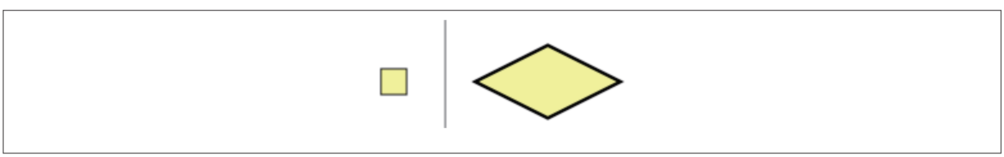


图 12-6：多个 <animateTransform> 之前和之后

我们也可以用 additive="sum" 合并控制数值和颜色属性的动画元素的效果。如果动画使用 to 指定，则把它们加在一起会导致后续动画使用上一个动画的当前值作为它们的起点。如果动画使用 by 属性定义效果，或者同时使用 from 和 to，那么最终值将会是所有单个变化的总和。

12.10 <animateMotion>元素

我们可以通过在 <animateTransform> 元素中使用 translate 让对象沿着一条直线路径运动。然而，如果想要对象按照更复杂的路径运动，我们需要一系列变换动画定时一个接一个地执行。<animateMotion> 元素使得让对象沿着任意路径运动变得很容易，无论是直线还是一系列的重叠循环路径。

如果想要对直线运动使用 <animateMotion>，简单地设置 from 和 to 属性，然后给每个属性分配一对 (*x*, *y*) 坐标即可。这个坐标用于指定要移动形状的坐标系统中 (0,0) 点的位置，类似于 translate(x, y) 的工作原理。示例 12-14 展示了将一组圆和矩形从 (0,0) 移动到 (60,30) 的例子。

示例 12-14：线性路径动画

http://oreillymedia.github.io/svg-essentials-examples/ch12/linear_animateMotion.html

```
<g>
  <rect x="0" y="0" width="30" height="30" style="fill: #ccc;"/>
  <circle cx="30" cy="30" r="15" style="fill: #cfc; stroke: green;" />
  <animateMotion from="0,0" to="60,30" dur="4s" fill="freeze"/>
</g>
```

可以用 values 属性指定多个点，但是运动仍然是一系列的直线运动。如果想要跟随更复杂的路径运动，要使用 path 属性；它的值与 <path> 元素的 d 属性的值格式相同。示例 12-15 改编自 SVG 规范，是一个按照三次贝塞尔曲线路径运动的三角形。

示例 12-15：沿复杂的路径运动

http://oreillymedia.github.io/svg-essentials-examples/ch12/complex_animate_motion.html

```
<!-- 要移动的三角形路径 -->
<path d="M50,125 C 100,25 150,225, 200, 125"
  style="fill: none; stroke: blue;"/>

<!-- 三角形会沿着运动路径移动。路径原点与三角形底边中点垂直对齐 -->
<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;">
  <animateMotion
    path="M50,125 C 100,25 150,225, 200, 125"
    dur="6s" fill="freeze"/>
</path>
```

正如我们在图 12-7 中看到的，三角形在整个路径中都保持垂直方向。

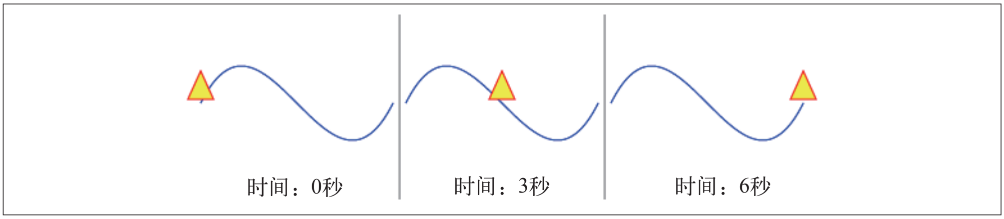


图 12-7：沿着复杂路径的 <animateMotion>

如果希望对象倾斜以使其 *x* 轴始终平行于路径的方向，只要给 <animateMotion> 元素添加一个值为 auto 的 rotate 属性就行了。示例 12-16 展示了这个 SVG，图 12-8 展示了动画不同阶段的截屏。

示例 12-16：带自动旋转的复杂路径动画

http://oreillymedia.github.io/svg-essentials-examples/ch12/animate_motion_rotate.html

```
<!-- 要移动的三角形路径 -->
<path d="M50,125 C 100,25 150,225, 200, 125"
  style="fill: none; stroke: blue;"/>

<!-- 三角形会沿着运动路径移动。它通过在原点上方垂直方向上与三角形底边中心水平对齐的
方式定义 -->
<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;">
  <animateMotion
    path="M50,125 C 100,25 150,225, 200, 125"
    rotate="auto"
    dur="6s" fill="freeze"/>
</path>
```

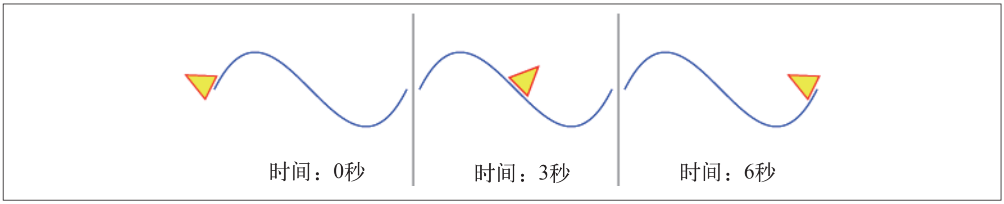


图 12-8：沿着带自动旋转的复杂路径的 <animateMotion>

简而言之，当我们不使用 rotate 属性时，其默认值为 0，此时对象就像一个沿着路径浮动的热气球。如果设置 rotate 为 auto，对象就像坐过山车一样，沿着路行向上或向下倾斜。

我们也可以设置 roate 为一个数值，这会使对象在动画中旋转。因此，如果希望无论路径是什么方向，对象都旋转 45 度，就使用 rotate="45"。

示例 12-16 使用蓝色绘制了路径，因此它是可见的，然后在 <animateMotion> 元素中复制了该路径。要想避免复制，只需在 <animateMotion> 元素中添加一个 <mpath> 元素。<mpath> 将会包含一个 xlink:href 属性引用我们想要使用的路径。当想要将一个路径应用给多个动

画对象时，这也可以派上用场。示例 12-17 使用 <mpath> 重写了前面的例子。

示例 12-17：使用 <mpath> 定义沿着复杂路径运动的动画

```
<path id="cubicCurve" d="M50,125 C 100,25 150,225, 200, 125"
  style="fill: none; stroke: blue;"/>

<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;">
  <animateMotion dur="6s" rotate="auto" fill="freeze">
    <mpath xlink:href="#cubicCurve"/>
  </animateMotion>
</path>
```

12.11　为运动指定关键点和时间

在示例 12-16 中，三角形是匀速移动的。匀速动画是 <animateMotion> 的默认行为（等价于 calcMode="paced"）；后续移动之间需要花费的时间与它们之间的距离成正比。[2]

在 12.7 节，我们介绍了 keyTimes 属性，它可以用来控制动画在不同值之间过渡的速度。我们也可以针对运动动画使用 keyTimes，但是如果我们使用路径而不是 value 列表定义运动，那么需要针对路径使用 keyPoints 属性指定关键点（你肯定猜到了）。

和 keyTimes 一样，keyPoints 是一个分号分隔的十进制数值列表。每个点表示对象应该按照 keyTimes 列表中相应的时间点沿着路径移动多远。正如 keyTimes 的范围是从 0（动画起始）到 1（动画结束），keyPoints 的范围也是从 0（路径的开始）到 1（路径结束）。示例 12-18 中的三角形在上升时越来越慢。

虽然 keyTimes 必须按照从 0 到 1 的顺序指定，但 keyPoints 可以在路径的中间开始或结束，还可以向两个方向运动。然而，keyPoints 和 keyTimes 列表必须拥有相同数量的条目，并且我们必须设置 calcMode="linear"（或者 "spline"，但这不在本书讨论范围之内）。

示例 12-18：使用 keyPoints 和 keyTimes 沿着路径做变速运动

http://oreillymedia.github.io/svg-essentials-examples/ch12/key_points.html

```
<path d="M-10,-3 L10,-3 L0,-25z" style="fill: yellow; stroke: red;" >
  <animateMotion
    path="M50,125 C 100,25 150,225, 200, 125"
    rotate="auto"
    keyPoints="0;0.2;0.8;1"
    keyTimes="0;0.33;0.66;1"
    calcMode="linear"
    dur="6s" fill="freeze"/>
</path>
```

注 2：对于速度和距离规则有一个例外：如果我们的路径含有任意的 moveto 命令，都会被算作零距离，意味着我们的对象将会从一个子路径的结束位置立即跳转到下一个子路径的开始位置。

12.12 使用CSS处理SVG动画

现代浏览器允许我们使用 CSS 处理 HTML 和 SVG 动画。这需要两个步骤。第一步，选中想要运动的元素，然后设置将动画属性作为一个整体进行设置。第二步，告诉浏览器改变选中元素的哪个属性以及在动画的什么阶段；这些都定义在 `@keyframes` 说明符中。

考虑下面这个任务：显示一个绿色的星星，使其逐渐变淡为白色的内部，同时让边框变粗；这个效果就是颜色“流出”到边框。下面是绘制该星星的 SVG：

```
<svg width="200" height="200" viewBox="0 0 200 200">
  <defs>
    <g id="starDef">
      <path d="M 38.042 -12.361 9.405 -12.944 -0.000 -40.000
        -9.405 -12.944 -38.042 -12.361 -15.217 4.944
        -23.511 32.361 0.000 16.000 23.511 32.361 15.217 4.944 Z"/>
    </g>
  </defs>
  <use id="star" class="starStyle" xlink:href="#starDef"
    transform="translate(100, 100)"
    style="fill: #008000; stroke: #008000"/>
</svg>
```

12.12.1 动画属性

下面是我们可以在 CSS 中给动画元素设置的属性。

- `animation-name` 就是 `@keyframes` 说明符的名称。
- `animation-duration` 决定动画应该持续多久；它的值是数字后面紧跟一个 12.2 节中所描述的时间单位。
- `animation-timing-function` 告诉浏览器如何计算中间值（比如,动画应该缓入或者缓出，还是按照不连续的步骤运动）。
- `animation-iteration-count` 告诉浏览器重复动画多少次，`infinite` 表示无限循环。
- `animation-direction` 决定动画是反向还是正向执行，以及是否在两个值之间交替。
- `animation-play-state` 可以设置为 `running` 或者 `paused`。
- `animation-delay` 告诉浏览器应用样式之后等待多久才开始动画。
- `animation-fill-mode` 告诉动画不再执行时使用什么属性。可以是 `forwards`（应用动画结束时的属性值）、`backward`（应用动画开始时的属性值）或者 `both`。

> 编写本文时，如果要在基于 Webkit 的浏览器中使用这些属性，我们必须针对它们使用 `-webkit-` 前缀，例如，`-webkit-animation-name` 或者 `-webkit-animation-duration`。

示例 12-19 展示了用 CSS 设置星星动画。它会重复 4 次，每次动画时长 2 秒。

示例 12-19：使用 CSS 设置动画

```
.starStyle {
  animation-name: starAnim;
  animation-duration: 2s;
  animation-iteration-count: 4;
  animation-direction: alternate;
  animation-timing-function: ease;
  animation-play-state: running;
}
```

12.12.2　设置动画关键帧

我们可以通过使用 `@keyframes` 这一媒体类型设置动画每个阶段要改变的属性，后面紧跟一个控制动画的名称。在 `@keyframes` 内，我们要列出 keyframe 选择器，即告诉浏览器何时改变属性的百分比。对于每个选择器，我们要列出动画应该呈现的属性和值。示例 12-20 展示了星星动画的关键帧。对于基于 Webit 的浏览器，请使用 `@-webkit-keyframes`。图 12-9 展示了动画的三个阶段。

示例 12-20 ：CSS 关键帧规范

http://oreillymedia.github.io/svg-essentials-examples/ch12/svg_css_anim1.html

```
@keyframes starAnim {
  0% {
    fill-opacity: 1.0;
    stroke-width: 0;
  }

  100% {
    fill-opacity: 0;
    stroke-width: 6;
  }
}
```

也可以使用 0% 和 100% 的同义词 `from` 和 `to`。



图 12-9：开始、中间和结束阶段的动画

12.12.3 CSS中的动画运动

如果想要使用纯 CSS 处理动画运动，我们不能使用 `transform` 属性。相反，必须使用 CSS 样式平移、旋转和缩放 SVG。幸运的是，正如 6.9 节中指出的，尽管 CSS `transform` 属性的值和 SVG `transform` 属性的值有些差异，但是看起来还是非常像的。如果想要将 SVG 元素平移到 (100,50)，缩放 1.5 倍，然后旋转 90 度，属性将会是如下所示：

```
transform: translate(100px, 50px) scale(1.5) rotate(90deg);
```

示例 12-21 展示了让星星向上移动并旋转，然后降落到起点所需要的关键帧。由于 100% 关键帧没有指定 `translate`，星星会回到它的原始位置（指定在 SVG 中）。这也是除了 20% 这一关键帧，50% 和 80% 关键帧必须指定 `translate` 的原因，有了它，星星在该部分动画期间才不会垂直移动。

示例 12-21：在 CSS 中指定变换

```
@keyframes starAnim {
  0% {
    fill-opacity: 1.0;
    stroke-width: 0;
  }

  20% {
    transform: translate(100px, 50px)
  }

  50% {
    transform: translate(100px, 50px) rotate(180deg)
  }

  80% {
    transform: translate(100px, 50px) rotate(360deg)
  }

  100% {
    fill-opacity: 0.0;
    stroke-width: 6;
  }
}
```

在线示例中我们可以创建自己的关键帧并实验 timing 属性：

http://oreillymedia.github.io/svg-essentials-examples/ch12/svg_css_anim2.html

> 编写本文时，当我们给使用 `<use>` 元素复制的 SVG 图形应用 CSS 动画和过渡时，浏览器还有一些缺陷且表现不一致。

第13章

添加交互

到目前为止，作为 SVG 文档的作者，你已经做完了关于图形的所有决定，比如静态图像看起来是什么样子，如果有动画的话何时启动和停止等。本章，我们将了解到如何将部分控制权移交给正在查看图像的人。

最低级的交互方式是*声明式交互*（declarative interactivity），即定义一些动画或者样式改动，告诉浏览器在某些情况下启用这些动画或者样式改动，而不是通过脚本去控制。SVG 提供了有限的内置交互状态。

13.1　在SVG中使用链接

最简单的交互方式是由 `<a>` 元素实现的链接。将图形包含在 `<a>` 元素内，这个图形就具有链接了；点击图形后，将跳转至 `xlink:href` 属性指定的 URL。你可以链接到另一个 SVG 文件，或者在条件允许的情况下链接到一个网页。在示例 13-1 中，点击“Cat”时将链接到一个画有一只猫的 SVG 文件；当点击红色、绿色和蓝色图形时会链接到 W3C 的 SVG 规范页面。第二个链接中的所有图形分别链接到了相同的目的地址，而不是整个图形边框中的区域都会响应。当我们测试这个例子，在图形之间移动光标时，会发现这些图形之间的区域不能响应点击。

示例 13-1：SVG 中的链接

http://oreillymedia.github.io/svg-essentials-examples/ch13/svg_link.svg

```
<a xlink:href="cat.svg">
  <text x="100" y="30" style="font-size: 12pt;">Cat</text>
```

```
</a>

<a xlink:href="http://www.w3.org/SVG/">
  <circle cx="50" cy="70" r="20" style="fill: red;"/>
  <rect x="75" y="50" width="40" height="40" style="fill: green;"/>
  <path d="M120 90, 140 50, 160 90 Z" style="fill: blue;"/>
</a>
```

如 5.3.2 节所描述的，在 `<use>` 元素中，`xlink:href` 指定一个资源，它会成为图形的一部分。对于 `<a>` 元素，`xlink:href` 属性指定的是另一个用于跳转的资源。

HTML 文档中的链接可以通过颜色和下划线效果来识别。图 13-1 展示了示例 13-1 的结果；没有任何地方告诉我们图形是有链接的，除非我们注意到光标从箭头更改为“手形”。使用键盘操作的用户也会面临同样的困难：某些浏览器会在被聚焦的元素上显示轮廓，但另一些却没有。我们可以使用 CSS 伪类向用户提供有关图形交互元素的一些反馈。和 CSS class 一样，伪类用于给元素的特定实例应用样式；但和真正的 class 不同的是，它们会自动生效，而不需要写在 `class` 属性中。[1]

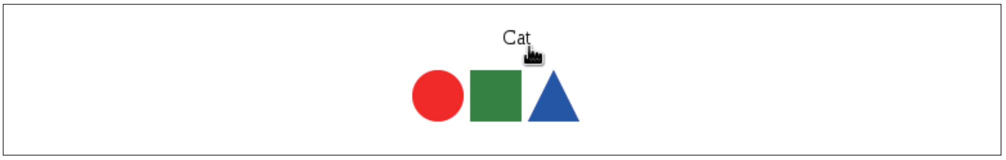


图 13-1：SVG 超链接，示例 13-1 的结果

当鼠标指针在元素上方时，`:hover` 伪类会生效，而当键盘操作聚焦到某个元素上时，`:focus` 伪类会生效。因为这两个伪类都指示了用户操作的可能性，所以我们经常对 `element:hover` 和 `element:focus` 应用相同的样式。

示例 13-2 使用了和示例 13-1 相同的图形，但当链接被鼠标指向或者作为键盘焦点时，会有反馈。文本将变成粗体加下划线，并且图形会有淡蓝色的边框。

示例 13-2：使用 CSS 高亮 SVG 链接

http://oreillymedia.github.io/svg-essentials-examples/ch13/svg_css_link.svg

```
<style type="text/css"><![CDATA[

    a.words:hover, a.words:focus {
       text-decoration: underline;
       font-weight: bold;
    }
  a.shapes:hover, a.shapes:focus {
     stroke: #66f;
     stroke-width: 2;
```

注 1：Apache Batik SVG 1.7 版阅读器不支持 CSS 伪类选择器。

```
    outline: none; /* 覆盖默认的聚焦样式 */
  }
  ]]>

</style>

<a class="words" xlink:href="cat.svg">
  <text x="100" y="30" style="font-size: 12pt;">Cat</text>
</a>

<a class="shapes" xlink:href="http://www.w3.org/SVG/">
  <circle cx="50" cy="70" r="20" style="fill: red;"/>
  <rect x="75" y="50" width="40" height="40" style="fill: green;"/>
  <path d="M120 90, 140 50, 160 90 Z" style="fill: blue;"/>
</a>
```

13.2　控制CSS动画

我们是否想要更加动态的用户反馈？ CSS 动画也使用样式属性定义，所以它们也可以由伪类控制。示例 13-3 中，当用鼠标悬停在图形上时，动画就会开始。

示例 13-3：使用 :hover 属性关联动画

http://oreillymedia.github.io/svg-essentials-examples/ch13/anim_css_link.svg

```
<style type="text/css"><![CDATA[

  a.animatedLink {
    animation-name: animKeys;
    animation-iteration-count: infinite;
    animation-duration: 0.5s;
    animation-direction: alternate;
    animation-play-state: paused;
  }

  a.animatedLink:hover {
    animation-play-state: running;
  }

  @keyframes animKeys {
    0% {fill-opacity: 1.0;}
    100% {fill-opacity: 0.5;}
  }
  ]]>

  </style>

<a class="animatedLink" xlink:href="http://www.w3.org/SVG/">
  <circle cx="50" cy="70" r="20" style="fill: red;"/>
  <rect x="75" y="50" width="40" height="40" style="fill: green;"/>
  <path d="M120 90, 140 50, 160 90 Z" style="fill: blue;"/>
</a>
```

从设计的角度看，这并不是一个很好的示例：如果动画正在进行时，将鼠标移出链接区域，动画仅仅会暂停，而不会恢复到完全不透明。如果你对 CSS 动画感兴趣，可以参考最新的规范草案：http://www.w3.org/TR/css3-animations/。

13.3 用户触发的SMIL动画

CSS 动画的局限在于只能用于有限的变动类型和有限的事件响应。如果我们在 SVG 中使用 SMIL 动画元素，可以为 `begin` 和 `end` 属性使用另一种格式的值，以便响应用户操作。

这种可以响应交互式动画时间属性的格式就是 `elementID.eventName`。通过 ID 引用的元素并不一定是正在进行动画的元素。

利用 SMIL 的 `begin` 和 `end` 属性进行交互的过程是基于事件的：一旦动画开始，它就会进行下去，直到完成动画或者动画终止事件发生。与此相反，使用 CSS 伪类的交互是基于状态的：样式或者动画只在状态为真时才会应用。例如，`#myElement:hover` 这个 CSS 伪类选择器描述了当鼠标指针悬停在 ID 为 `myElement` 的元素上时的状态；当需要定义在这个状态下产生的动画时，需要设置动画属性 `begin="myElement.mouseover"` 和 `end="myElement.mouseout"`。[2]

为了更好地控制动画，我们可以选择性地以 `elementID.eventName + offset` 的形式添加时间偏移。指定这个偏移的格式和指定其他 SMIL 动画时间属性一样（参见 12.2 节），比如带单位的 `1.5min`，或者和秒表计时一样，类似于 `01:30`。你甚至可以指定负的时间偏移，但因为没有什么超能力的矢量图形，所以动画并不会在事件发生前开始。但当事件发生的时候，动画会跳过前面的部分（即在事件发生时已经流逝的时间中发生的动画），继续进行剩下的动画。

示例 13-4 创建了一个不规则图形和一个按钮。当你点击按钮的时候，不规则图形会旋转 360 度。点击前和点击后的截图见图 13-2。

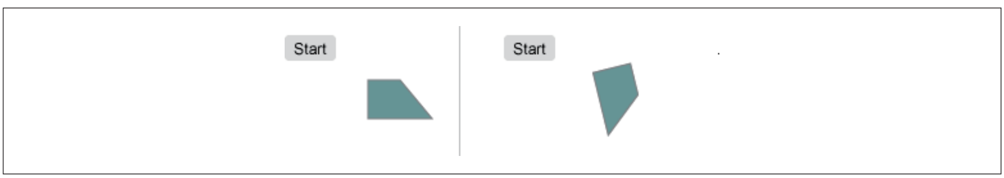


图 13-2：脚本控制的动画的两个阶段的截图

示例 13-4：交互式动画

http://oreillymedia.github.io/svg-essentials-examples/ch13/smil_event_animation.svg

注 2：如果你熟悉 JavaScript 事件处理的话，会发现这些事件名称和 DOM 事件处理一样。如果你不熟悉 JavaScript 和 DOM 事件，请继续阅读。

```
<g id="button">     ❶
  <rect x="10" y="10" width="40" height="20" rx="4" ry="4"
    style="fill: #ddd;"/>
  <text x="30" y="25"
    style="text-anchor: middle; font-size: 8pt">Start</text>
</g>

<g transform="translate(100, 60)">
  <path d="M-25 -15, 0 -15, 25 15, -25 15 Z"
    style="stroke: gray; fill: #699;">

    <animateTransform id="trapezoid" attributeName="transform"
      type="rotate" from="0" to="360"
      begin="button.click"
      dur="6s"/>   ❷
  </path>
</g>
```

❶ 开始按钮只是简单地使用了圆角矩形和文本。id 写在整个组上。

❷ 我们在 button 对象中绑定了 click 事件，点击时动画开始，而没有使用以秒为单位的开始时间来控制动画。

> 通常情况下应该先设计 SVG，再添加脚本。这种方法的一个好处是，在添加交互之前可以先看看图形是否符合要求。

13.4　使用脚本控制SVG

前面我们使用的是声明式动画，下一步我们将换成脚本来处理动画，这也是一大步改变。你可以使用 ECMAScript 编程来与 SVG 图形进行交互。[ECMAScript 是 JavaScript 标准化版本，由 ECMA 组织（European Computer Manufacturer's Association，欧洲计算机制造联合会）定义。] 如果你没有接触过 ECMA/JavaScript，或者没有接触过编程，可以参考附录 C。

SVG 阅读器在读取 SVG 文档的标记时，会创建一棵节点树，也就是内存中的一些对象，它们和标记的结构、内容一一对应。这就是 DOM（Document Object Model，文档对象模型），你可以通过脚本来访问。

在你处理 DOM 之前，首先要做的就是获取到节点。最主要的方法就是 document.getElementById(idString)。这个方法接受一个字符串类型的参数，代表 SVG 元素的 id，返回 DOM 中对应节点的引用。如果你想要获取文档中标记名（标记名指 <svg> 中的 svg，<rect> 中的 rect 等）为某个名字的所有元素，可以使用 document 上的另一个方法

getElementsByTagName(name)，它返回一个节点数组。

当你获取到元素的节点之后，就可以：

- 使用 element.getAttribute(attributeName) 读取它的属性,以字符串形式返回属性的值；
- 使用 element.setAttribute(name, newValue) 改变属性值，如果指定名称的属性不存在，则会创建；
- 使用 element.removeAttribute(name) 删除属性。

你可以使用 element.setAttribute("style", newStyleValue) 来修改内联样式，但这样会覆盖元素上的所有样式。为了防止样式被覆盖，你可以使用 element.style 属性。

- element.style.getPropertyValue(propertyName) 获取指定样式。
- element.style.setProperty(propertyName, newValue, priority) 修改属性（priority 通常为 null，但也可以是 important）。
- element.style.removeProperty(propertyName) 删除属性。

如果你真的希望一次设置所有的样式，也可以直接修改 element.style.cssText，这个属性是一个代表所有样式的字符串，格式为 property-name: value。[3]

如果你需要获取或者修改节点的文本内容，使用 element.textContent 属性。读取该属性时，它会返回节点所有后代的文本拼接后的字符串。修改该属性时，则会用一个文本块替换所有的后代节点。[4]

示例 13-5 使用了这些方法来获取 SVG 元素的属性，以文本方式显示它们，然后修改某个属性。（这个例子还没有交互，但对我们来说很重要，因为它是更多交互的基础。）

示例 13-5：使用 DOM 获取 SVG

http://oreillymedia.github.io/svg-essentials-examples/ch13/basic_dom_example.svg

```
<svg width="300" height="100" viewBox="0 0 300 100"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink">

  <title>Accessing Content in SVG</title>

  <rect id="rectangle" x="10" y="20" width="30" height="40"
    style="stroke:gray; fill: #ff9; stroke-width:3"/> ❶
  <text id="output" x="10" y="80" style="font-size:9pt"></text>
```

注 3：大部分浏览器允许你通过 element.style.propertyName 或者 element.style["property-name"] 读写样式属性，但这并不属于 CSS 对象模型标准（http://www.w3.org/TR/cssom/），在一些 SVG 阅读器中也不支持，或者支持的情况不一致。

注 4：如果你对使用 .innerHTML 属性来修改组合文本和标记所有子元素很熟悉的话，请注意，该属性只在 HTMLElement 中被定义，很多浏览器和 SVG 阅读器在 SVG 元素上不支持该方法。

```
  <script type="application/ecmascript">
  // <![CDATA[ ❷
    var txt = document.getElementById("output"); ❸
    var r = document.getElementById("rectangle");
    var msg = r.getAttribute("x") + ", " +  ❹
      r.getAttribute("y") + " " +
      r.style.getPropertyValue("stroke") + " " +
      r.style.getPropertyValue("fill");
    r.setAttribute("height", "30"); ❺
    txt.textContent= msg; ❻
    // ]]>
  </script>
</svg>
```

❶ 为了更容易从脚本中获取元素，给它一个唯一的 id。

❷ <![CDATA[用于保证代码中的 < 和 > 不被解析为标记。

❸ 从文档中通过 id 选择元素，并将结果保存在变量中。

❹ getAttribute() 和 style.getPropertyValue() 的结果是字符串，使用 + 将它们连接在一起构成消息字符串。

❺ 修改矩形的高度，让它变成正方形。

❻ 最后，设置 <text> 元素的内容来显示属性和属性值。

> 示例 13-5 中的脚本是在 <rect> 和 <text> 元素之后引入到 SVG 文件中的，这样可以保证在脚本运行之前，元素已经在文档中存在。

13.4.1 事件概览

当图形对象响应事件的时候就产生了交互。事件的类型有很多种。下方的事件描述有很多来自万维网联盟的规范（http://www.w3.org/TR/SVG/interact.html#SVGEvents）。

- 用户接口事件

  当元素接受焦点和失去焦点（比如选中和取消选中文本）时会分别触发 focusIn 和 focusOut 事件。当一个元素通过鼠标点击或者键盘操作激活时，会变成 activate 元素。

- 鼠标事件

  当使用指针设备（如鼠标）在某个元素上点击和释放时会分别触发 mousedown 和 mouseup 事件。如果这两个事件在屏幕上的位置相同，则会触发 click 事件。

  当指针设备移动进入某个元素、在元素内移动、从元素中移走时，会分别触发 mouseover、mousemove 和 mouseout 事件。

- DOM 变化（mutation）事件

  当 DOM（被其他脚本改动导致）变化时，SVG 阅读器会触发一些事件。比如，当一个节点被添加到另一个节点作为子节点时，会触发 DOMNodeInserted 事件；当节点的属性发生变化时，会触发 DOMAttrModified 事件。本书不会详细描述与这些事件有关的内容。

- 文档事件

  当 SVG 阅读器完全解析完文档，并准备好做进一步操作（比如显示到设备上）时，会触发 SVGLoad 事件。当文档被从窗口中移除时，会触发 SVGUnload 事件。当页面正在加载时被突然中止，会触发 SVGAbort 事件。当文档不能正确加载或者脚本执行有错误时，会触发 SVGError 事件。

  当阅读器文档的大小、滚动位置、缩放比例发生变化时，会触发 SVGResize、SVGScroll、SVGZoom 事件。

- 动画事件

  动画开始、结束和重复播放时会触发 beginEvent、endEvent 和 repeatEvent 事件。repeatEvent 在第一次播放时不会触发。

- 键盘事件

  SVG 并没有原生的键盘事件，但是有一些浏览器可能支持 keydown 和 keyup 事件，这是不标准的。

13.4.2　监听和响应事件

要让一个对象响应事件，首先需要让对象监听事件。我们通过 addEventListener() 函数来监听事件，它接受两个参数。第一个参数是表示要监听的事件类型的字符串。第二个参数是处理事件的函数。第三个参数是可选的，用来表示是否响应“事件捕获”阶段的事件。事件捕获是指阅读器将事件从根元素传递到子元素直到找到指定事件目标的过程。指定 false（通常用 false，也是默认值）表示响应“事件冒泡”阶段的事件。事件冒泡是指事件从触发事件的子元素一直传递到目标元素，再传递到根元素的过程。[5]

事件处理函数接受一个参数：一个包含了触发函数调用的事件的相关信息的事件对象。事件对象中最重要的属性是 target 属性，它代表事件产生的元素。还有一些重要的属性，比如 clientX 和 clientY，给出了事件发生的坐标，这个坐标是相对于整个 SVG 文件或者整个 Web 页面而言的。

示例 13-6 监听了圆上的 mouseover、mouseout 和 click 事件。鼠标移入移出圆会分别导致

注 5：这是一个很简单的定义，详情请参阅 DOM 事件标准（http://www.w3.org/TR/DOM-Level-3-Events/#event-flow）。

圆角半径的增和减，点击鼠标则会增或减圆的轮廓宽度。

示例 13-6：监听鼠标运动

http://oreillymedia.github.io/svg-essentials-examples/ch13/simple_event.svg

```
<circle id="circle" cx="50" cy="50" r="20"
  style="fill: #ff9; stroke:black; stroke-width: 1"/>

<script type="application/ecmascript"><![CDATA[

  function grow(evt) {
    var obj = evt.target;
    obj.setAttribute("r", "30");
  }

  function shrink(evt) {
    this.setAttribute("r", "20");
  }

  function reStroke(evt) {
    var w = evt.target.style.getPropertyValue("stroke-width");
    w = 4 - parseFloat(w); /* toggle between 1 and 3 */
    evt.target.style.setProperty("stroke-width", w, null);
  }

  var c = document.getElementById("circle");
  c.addEventListener("mouseover", grow);
  c.addEventListener("mouseout", shrink);
  c.addEventListener("click", reStroke);
  // ]]>

</script>
```

第一个事件处理函数 `grow()` 使用了 `evt.target` 来获取接收事件的元素，然后将它存在变量中。第二个事件处理函数 `shrink()` 使用了关键字 `this` 来引用跟事件监听函数绑定的元素（在这个例子中，和 `evt.target` 相同，但并不总是这样）。最后一个事件处理函数 `reStroke()` 也使用了 `evt.target`，但没有使用临时变量。

我们也可以使用 `c` 来代替 `evt.target`（因为这是唯一绑定事件处理函数的元素），但这是一种非常不好的办法，因为你经常需要将同一个事件处理函数绑定到不同的元素上，比如下一个例子。

13.4.3 修改多个对象的属性

有时候你会希望在对象 A 上产生的事件可以同时影响对象 A 和对象 B 的属性。示例 13-7 展示的可能是世界上最丑的电商网站了。在图 13-3 中展示了一件 T 恤，当用户点击选择尺码的时候，T 恤的尺寸也会随之改变。当前选中的尺码按钮会高亮成淡黄色背景。

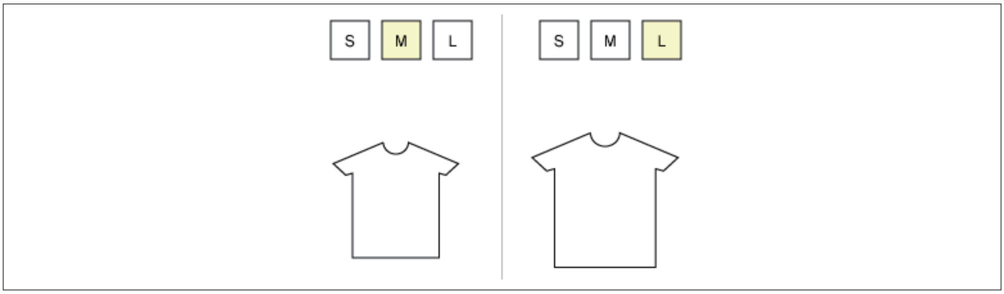


图 13-3：不同尺寸选择的截图

示例 13-7：使用脚本修改多个对象

http://oreillymedia.github.io/svg-essentials-examples/ch13/shirt1.svg

XML 代码：

```
<svg width="400" height="250" viewBox="0 0 400 250"
  xmlns="http://www.w3.org/2000/svg"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  onload="init(evt)"> ❶

  <defs>
    <style type="text/css" > <![CDATA[
      /* style rules will go here */
    ]]></style>
    <script type="application/ecmascript"> <![CDATA[
      /* script will go here */
    ]]></script>

    <path id="shirt-outline"
      d="M -6 -30 -32 -19 -25.5 -13 -22 -14 -22 30 23 30
        23 -14 26.5 -13 33 -19 7 -30
        A 6.5 6 0 0 1 -6 -30"/> ❷
  </defs>

  <g id="shirt" >
    <use xlink:href="#shirt-outline" x="0" y="0"/>
  </g>

  <g id="scale0" >
    <rect x="100" y="10" width="30" height="30" />
    <text x="115" y="30">S</text>
  </g>

  <g id="scale1" class="selected"> ❸
    <rect x="140" y="10" width="30" height="30" />
    <text x="155" y="30">M</text>
  </g>

  <g id="scale2" >
    <rect x="180" y="10" width="30" height="30" />
```

```
    <text x="195" y="30">L</text>
  </g>
</svg>
```

❶ 当文档加载完成后，会触发 SVGLoad 事件，onload 函数会调用初始化函数，并传入事件信息。很多脚本都会这样处理，以确保所有的变量都被正确地初始化。这也使得你可以将 <script> 标记放到 SVG 元素之前，以方便维护。使用 oneventname 属性的形式也可以监听很多事件，但是不建议这样做（应该使用 addEventListener()），因为这样会将脚本函数和 XML 结构混合起来。不过文档加载事件是个例外。

❷ T 恤的轮廓中心是 (0,0)，然后通过变换移到对应的位置，所以它的缩放中心点就是 T 恤中心。

❸ 中间的按钮在初始化时被选中。

样式：

```
svg { /* 默认值 */
   stroke: black;
   fill: white;
}
g.selected rect {
   fill: #ffc; /* 淡黄色 */
}
text {
   stroke: none;
   fill:black;
   text-anchor: middle;
}
```

selected class 被用来标识哪个尺寸是被选中的，样式为淡黄色背景。

脚本：

```
var scaleChoice = 1;  ❶
var scaleFactor = [1.25, 1.5, 1.75];

function init(evt) { ❷
  var obj;
  for (var i = 0; i < 3; i++) {
    obj = document.getElementById("scale" + i);
    obj.addEventListener("click", clickButton, false);
  }
  transformShirt();
}

function clickButton(evt) {
  var choice = evt.target.parentNode; ❸
  var name = choice.getAttribute("id");
  var old = document.getElementById("scale" + scaleChoice);
  old.removeAttribute("class"); ❹
  choice.setAttribute("class", "selected");
```

```
    scaleChoice = parseInt(name[name.length - 1]); ❺
    transformShirt();
  }

  function transformShirt() { ❻
    var factor = scaleFactor[scaleChoice];
    var obj = document.getElementById("shirt");
    obj.setAttribute("transform",
      "translate(150, 150) " +
      "scale(" + factor + ")");
    obj.setAttribute("stroke-width",
      1 / factor);
  }
```

❶ 这段脚本会关注哪个按钮（S、M、L）被选择了，然后根据索引从 `scaleFactor` 数组中取到对应的缩放值。默认的索引为 1，代表中号（M）。

❷ `init()` 函数会获取每个由矩形和文本组成的 `<g>`，并让它们监听 `click` 事件。然后在事件处理函数中让 T 恤显示正确的大小。

❸ 点击事件可能发生在文本上，也可能发生在矩形中。使用 `parentNode` 可以保证 `choice` 代表的是 `<g>` 对象。

❹ 将之前尺寸对应的按钮 `<g>` 上的 `selected` class 移除掉，然后为新选择的 `<g>` 添加这个 class。

❺ 通过按钮 `<g>` 的 `id` 的后半段提取出数字，赋值给 `scaleChoice`。这是个全局变量，稍后可以被 `transformShirt()` 函数获取到。

❻ 改变 T 恤的大小和位置是通过设置它的 `transform` 属性完成的。轮廓的宽度则是通过取缩放值的倒数来指定，这样可以保证当 T 恤放大和缩小时，轮廓在视觉上是一样宽的。

13.4.4　拖拽对象

我们来扩展这个示例，添加一些可以拖动的滑块，来设置 T 恤的颜色。如图 13-4 所示。

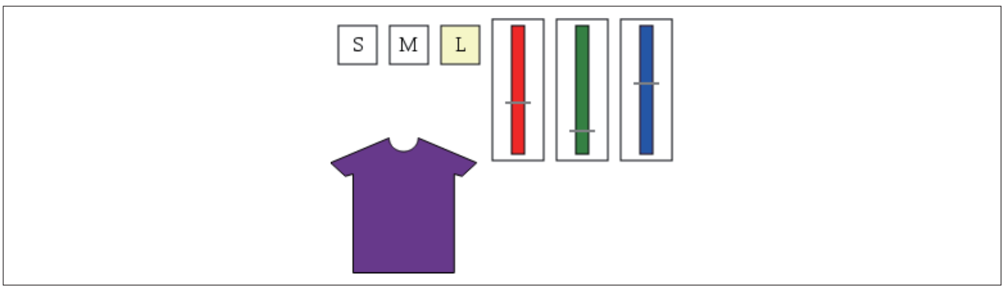


图 13-4：颜色滑块截图

你可以在线上示例中体验这些滑块：

http://oreillymedia.github.io/svg-essentials-examples/ch13/drag_objects.svg

这个示例的脚本需要更多的全局变量。首先是 slideChoice，用于标识当前被拖动的是哪一个滑块（0、1、2）。它的初始值是 -1，表示当前没有滑块被拖动。还需要使用一个 rgb 数组变量来分别保存红、绿、蓝的值（百分比），初始值都是 100，因为 T 恤一开始是白色的：

```
var slideChoice = -1;
var rgb = [100, 100, 100];
```

接下来绘制滑块。滑块的背景和上面指示当前值的线都是在白色背景上绘制的，然后将它们分组放在一起。id 属性放到 <line> 元素上，因为它的 y 坐标会变化。事件处理函数会被绑定到 <g> 上。这样这个组就会捕获发生在任何子元素上的鼠标事件。（这也是我们要画外面的白色框的原因；当鼠标移出背景外的时候，仍然可以跟踪到它的位置。）

```
<g id="sliderGroup0" transform="translate( 230, 10 )">
  <rect x="-10" y="-5" width="40" height="110"/>
  <rect x="5" y="0" width="10" height="100" style="fill: red;"/>
  <line id="slide0" class="slider"
    x1="0" y1="0" x2="20" y2="0" />
</g>

<g id="sliderGroup1" transform="translate( 280, 10 )">
  <rect x="-10" y="-5" width="40" height="110"/>
  <rect x="5" y="0" width="10" height="100" style="fill: green;"/>
  <line id="slide1" class="slider"
    x1="0" y1="0" x2="20" y2="0" />
</g>

<g id="sliderGroup2" transform="translate( 330, 10 )">
  <rect x="-10" y="-5" width="40" height="110"/>
  <rect x="5" y="0" width="10" height="100" style="fill: blue;"/>
  <line id="slide2" class="slider"
    x1="0" y1="0" x2="20" y2="0" />
</g>
```

新的样式规则包含了滑块所要的所有样式，只是没有指定它的颜色：

```
line.slider {
   stroke: gray;
   stroke-width: 2;
}
```

init() 函数分别为三个滑块添加了三个事件监听：

```
obj = document.getElementById("sliderGroup" + i);
obj.addEventListener("mousedown", startColorDrag, false);
obj.addEventListener("mousemove", doColorDrag, false);
obj.addEventListener("mouseup", endColorDrag, false);
```

对应的函数如下：

```
function startColorDrag(evt) { ❶
  var sliderId = evt.target.parentNode.getAttribute("id");
  endColorDrag( evt );
  slideChoice = parseInt(sliderId[sliderId.length - 1]);
}

function endColorDrag(evt) { ❷
  slideChoice = -1;
}

function doColorDrag(evt) { ❸
  var sliderId = evt.target.parentNode.getAttribute("id");
  chosen = parseInt(sliderId[sliderId.length - 1]);

  if (slideChoice >= 0 && slideChoice == chosen) {  ❹

    var obj = evt.target; ❺
    var pos = evt.clientY - 10;
    if (pos < 0) { pos = 0; }
    if (pos > 100) { pos = 100; }

    obj = document.getElementById("slide" + slideChoice); ❻
    obj.setAttribute("y1", pos);
    obj.setAttribute("y2", pos);

    rgb[slideChoice] = 100-pos; ❼

    var colorStr = "rgb(" + rgb[0] + "%," +  ❽
      rgb[1] + "%," + rgb[2] + "%)";
    obj = document.getElementById("shirt");
    obj.style.setProperty("fill", colorStr, null);
  }
}
```

❶ 鼠标按下的时候会调用 `startColorDrag(evt)`。它首先停止当前正在进行的拖动（如果有的话），然后将当前滑块设置为点击的这个（0= 红色，1= 绿色，2= 蓝色）。

❷ 鼠标按键释放的时候会调用 `endColorDrag(evt)`。这个函数也可能被其他函数调用。它会将当前滑块索引设为 -1，表示当前没有滑块被拖动。这个函数不需要处理 `evt` 参数。

❸ 鼠标移动时会调用 `doColorDrag(evt)`。这个函数会使用 `evt` 参数的 `target` 属性（来获知被拖动的是哪个滑块）和 `clientY` 属性（来获知当前鼠标的位置）。

❹ 检查当前是否有滑块在滑动，以及滑动的滑块是当前滑块。

❺ 获取指示线对象和鼠标位置（相对颜色块顶部换算过的位置），将位置值转换到 0~100 的范围。

❻ 将指示线移到新的鼠标位置。

❼ 计算该滑块对应的新颜色值。

❽ 以 `rgb()` 的形式写上颜色值以改变 T 恤的颜色。

这个例子中只有一个小地方需要特别注意：文档只会在鼠标指针在滑块内时响应 onmouseup 事件。所以如果你在红色的滑块上按下鼠标，然后拖动鼠标到 T 恤上，再释放鼠标，文档并不会注意到这个释放动作。当你把鼠标再次移回红色滑块时，它仍然会跟随鼠标。为了解决这个问题，我们添加了一个完全覆盖当前可视区域的透明矩形，然后让它来响应鼠标释放事件。当它响应 mouseup 事件时，会调用 stopColorDrag 方法。这个元素是第一个元素，这样它就会出现在图形的最底下。为了让这个元素不干扰图形，我们为它设置 style="fill: none;"。也许你会说："等等，不是说好了透明元素不响应事件的吗？"通常情况下的确如此，但我们可以将 pointer-events 的值设为 visible，这样不管它的透明度是多少，只要没被隐藏就能响应事件。[6]

```
<rect id="eventCatcher" x="0" y="0" width="400" height="300"
  style="fill: none;" pointer-events="visible" />
```

修改 init() 函数中的事件监听部分：

```
document.getElementById("eventCatcher").
  addEventListener("mouseup", endColorDrag, false);
```

13.4.5　与HTML页面交互

在第 2 章中，我们介绍过，在 HTML 文档中加入交互式的 SVG 有两种办法。如果只有少量 SVG，可以直接放入 HTML 中。如果有大量的 SVG 图形，可以通过 <object> 元素引用。下面的示例会展示如何将前面的例子放入 HTML 中：

```
<object id="externalShirt" data="shirt_interact.svg"
  type="image/svg+xml">
  <p>Alas, your browser does not support SVG.</p>
</object>
```

值得关注的是用于指定图形源（本例中是一个 URL，即一个相对路径）的 data 属性以及 type 属性，type 属性的值为 image/svg+xml。HTML 开启标记和结束标记之间的文字只会在图形无法加载时才显示出来。现在可以添加一些让 SVG 脚本和页面脚本交互的代码了，id 属性会在交互中扮演重要角色。

我们在 Web 页面中放置一个表单，让用户输入红色、绿色和蓝色的百分比。用户输入的值会反映到滑块上。如果用户拖动了滑块，那么表单中的值也会对应更新。

下面是 HTML 文档，其中引用了（还不存在的）updateSVG() 函数。这个函数会接受两个参数，分别是对应的滑块索引值和输入框中的值：

注 6：pointer-events 的其他值包括只响应填充区域（fill）、只响应轮廓区域（stroke）以及响应填充和轮廓区域（painted），这些值都不管元素是否可见。与之对应的还有 visibleFill、visibleStroke 和 visiblePainted，也会考虑元素的可见性。

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <title>SVG and HTML</title>
  <style type="text/css">
    /* 让表单项独占一行 */
    label {display: block;}
    h1 {font-size: 125%;}
  </style>
  <script type="text/javascript">
    /* 这里放置脚本代码 */
  </script>
</head>

<body>
<h1>SVG and HTML</h1>
<div style="text-align:center">
  <object id="shirt" data="shirt_interact.svg"
    type="image/svg+xml">
    <p>Alas, your browser cannot load this SVG file.</p>
  </object>

  <form id="rgbForm">
    <label>Red: <input id="fld0" type="text" size="5" value="100"
      onchange="updateSVG(0, this.value)" />% </label>
    <label>Green: <input id="fld1" type="text" size="5" value="100"
      onchange="updateSVG(1, this.value)" />% </label>
    <label>Blue: <input id="fld2" type="text" size="5" value="100"
      onchange="updateSVG(2, this.value)" />%</label>
  </form>
</div>
</body>
</html>
```

为了避免演示代码过于复杂，我们没太注意编码风格，并且将脚本直接放入了 HTML 中。onchange="updateSVG(0, this.value)" 和添加一个 <input> 元素 change 事件监听器效果一样。当输入框的值变化时，执行 updateSVG(0, this.value)。

updateHTMLField 函数将由 SVG 文档的脚本来调用。它接受两个参数，一个是输入框的索引，一个是值，值会显示在对应的输入框中：

```
function updateSVG(which, amount) {
  amount = parseInt(amount);
  if (!isNaN(amount) && window.setShirtColor) {
    window.setShirtColor(which, amount);
  }
}

function updateHTMLField(which, percent) {
```

```
    document.getElementById("fld" + which).value = percent;
  }
```

接下来需要修改 SVG 文档。现在有两种方法修改 T 恤的颜色：一种是从滑块中取值，一种是从表单中取值。这样的话，我们首先需要将设置 T 恤颜色的代码从滑块拖动的代码中分离出来。修改后的 `doColorDrag(evt)` 函数会检测当前滑块并计算滑块的位置，但具体的修改会调用一个新方法 `svgSetShirtColor`：

```
function doColorDrag(evt) {
  if (slideChoice >= 0) {
    var sliderId = evt.target.parentNode.getAttribute("id");
    chosen = parseInt(sliderId[sliderId.length - 1]);
    if (slideChoice == chosen) {
      svgSetShirtColor(slideChoice, 100 - (evt.clientY - 10));
    }
  }
}
```

`svgSetShirtColor` 函数会处理之前 `doColorDrag` 做的部分工作，但有两点不同：一是它会使用传入的滑块索引作为第一个参数，而不再使用全局变量 `slideChoice`；二是值也作为参数传入。类似这样的代码变化是在将一些临时的演示代码转变为更模块化的代码时必须做的。

```
function svgSetShirtColor(which, percent) {
  var obj;
  var colorStr;
  var newText;

  if (percent < 0) { percent = 0; } ❶
  if (percent > 100) { percent = 100; }

  obj = document.getElementById("slide" + which); ❷
  obj.setAttribute("y1", 100 - percent);
  obj.setAttribute("y2", 100 - percent);
  rgb[which] = percent;

  colorStr = "rgb(" + rgb[0] + "%," + ❸
    rgb[1] + "%," + rgb[2] + "%)";
  obj = document.getElementById("shirt");
  obj.style.setProperty("fill", colorStr, null);
}
```

❶ 需要检查值是否在正确的范围。

❷ 滑块移动到了新位置。

❸ 改变 T 恤颜色的代码和之前一样。

接下来，在 `init` 函数中使用 `parent` 来将 SVG 文档的 `svgSetShirtColor` 函数赋值给 HTML 页面的 `setShirtColor` 变量。之所以可以这样做，是因为当一个文档被嵌入到另一个文档中时，子文档的全局变量 `parent` 会指向父文档的 `window` 对象。因为 `setShirtColor` 会成为

Web 页面中 window 的属性，所以页面可以访问它。接下来的代码完成了 HTML 到 SVG 的通信。在使用 parent 变量前，先测试一下，以确保它存在（即确认 SVG 确实是被内嵌在另一个文档中）。

```
function init( evt ) {
  // 添加事件监听
  if (parent) {
        parent.setShirtColor = svgSetShirtColor;
   }
  transformShirt();
}
```

最后一步是完成 SVG 到 HTML 的通信，以便用户使用滑块调整颜色时也能正常工作。我们在用户拖拽结束时更新 HTML 表单，而不是一直不停地更新。在 endColorDrag 函数中添加下面加粗的代码。如果滑块被拖动了，会调用父页面中的 updateHTMLfield 函数（如果存在的话），并带上滑块的索引和值。

```
function endColorDrag( ) {

  if (slideChoice >= 0) {
    if (parent)
      parent.updateHTMLField(slideChoice, rgb[slideChoice]);
  }

  // 标记当前没有正被拖动的滑块
  slideChoice = -1;
}
```

结果如图 13-5 所示。为了节省屏幕空间，截图被编辑过。

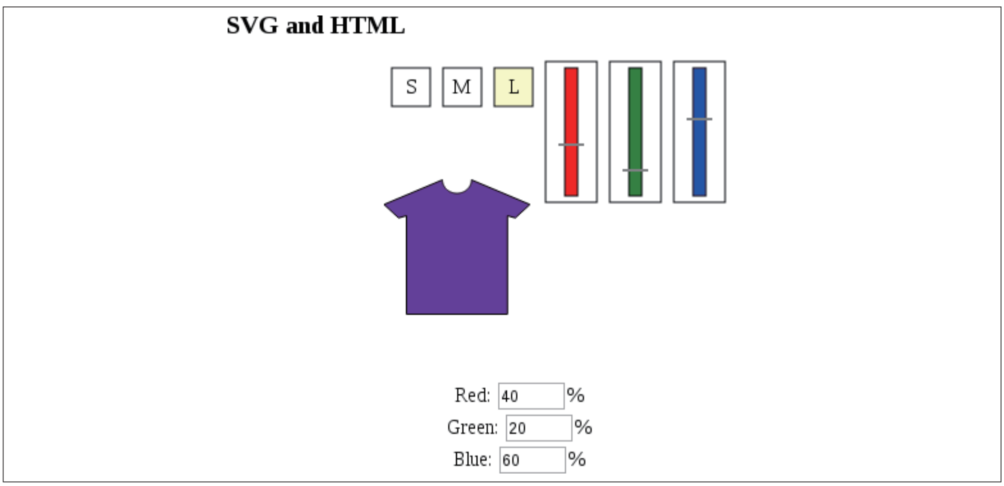


图 13-5：HTML 与 SVG 交互的截图

你也可以查看线上示例：

http://oreillymedia.github.io/svg-essentials-examples/ch13/shirt_interact.html

13.4.6 创建新元素

除了可以修改已有元素的属性外，脚本也可以创建新元素。接下来我们为 T 恤这个例子添加一些飞镖盘状的圆环。结果如图 13-6 所示，我们确信这会是一件超级流行的衣服。

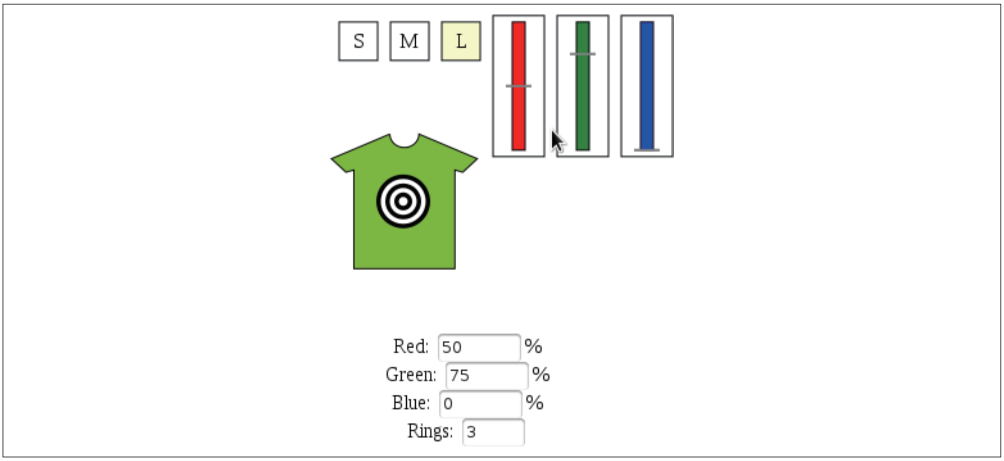


图 13-6：不同选项下的截图

你可以在在线示例中自己设计 T 恤：

http://oreillymedia.github.io/svg-essentials-examples/ch13/shirt_create.html

HTML 也需要做一些修改，添加一个新表单项来指定圆环的数量：

```
<label>Rings: <input id="nRings" type="text" size="3" value="0"
  onchange="createRings(this.value)" /></label>
```

SVG 图形在一个外部文档中，所以 HTML 中的脚本需要找到获取 SVG 图形的方法。在上一节中，SVG 文档中的脚本使用了 `parent` 来访问 HTML 脚本的环境。在这个示例中，HTML 中的脚本将使用 `<object>` 元素的 `getSVGDocument()` 方法来直接获取和修改 SVG DOM。SVG 文档和上一节完全一样。

为了访问到 SVG 文档，当页面加载完成时在 Web 页面的 `<body>` 上调用一个初始化方法：

```
<body onload="init()">
```

`init()` 函数会获取 SVG 文档并将它存在一个全局变量中：

```
var svgDoc;

function init() {
  var obj = document.getElementById("shirt");
  svgDoc = obj.getSVGDocument();
}
```

至此，HTML 页面脚本中的 createRings() 就能添加和移除 SVG 文档中的元素了，见示例 13-8。

示例 13-8：使用 JavaScript 创建元素

```
function createRings(nRings) {
  var shirt = svgDoc.getElementById("shirt"); ❶
  var rings = shirt.getElementsByTagName("circle"); ❷
  var i;
  var radius;
  var circle;

  for (i = rings.length - 1; i >= 0; i--) { ❸
    shirt.removeChild(rings[i]);
  }

  /* 限定范围为0-5 */
  if (nRings < 0) { nRings = 0; }
  else if (nRings > 5) { nRings = 5; }

  radius = nRings * 4;
  for (i = 0; i < nRings * 2; i++) {
    circle = svgDoc.createElementNS("http://www.w3.org/2000/svg",  ❹
      "circle");
    circle.setAttribute("cx", "0");
    circle.setAttribute("cy", "0");
    circle.setAttribute("r", radius);

    if (i % 2 == 0) { ❺
        circle.style.cssText = "fill:black; stroke:none";
    }
    else {
        circle.style.cssText = "fill:white; stroke:none;";
    }
    shirt.appendChild(circle); ❻
    radius -= 2;
  }
}
```

❶ 获取 SVG 文档中的 <g>。

❷ 获取组中的所有 <circle> 元素。

❸ 通过调用 shirt 变量所指向的父元素 <g> 的 removeChild(nodeToRemove)，（逆序）移除所有的圆环。

❹ 要创建的元素属于 SVG 文档，所以必须在 SVG 的命名空间（NS）中创建。注意命名空间是使用 URL 定义的，而不是前缀。

❺ 为奇偶位置的圆环设置不同的样式。因为我们是在为一个新的没有样式的元素设置多个样式，所以直接使用 `element.style.cssText` 设置。

❻ 最后，使用 `shirt` 分组的 `addChild(newNode)` 将新创建的 `<circle>` 放入文档中并减小圆角半径以便下个圆环使用。SVG 文档中的新节点会按照它们被添加的顺序排序，即从大到小。这样小圆环会出现在大圆环上方。

第 14 章

使用SVG DOM

在第 13 章的脚本中，我们已经使用过一些属性和方法，比如创建、选择元素，获取、设置属性和样式等，这些都是 DOM 标准的一部分，并不是 SVG 独有的。SVG1.1 规范为 SVG 定义了很多额外的方法，使得操作二维图形更容易。利用这些方法很容易精确指定文本或路径元素的位置，控制动画时间，以及在不同坐标系统之间自由转换。

14.1　确定元素的属性值

在 13.4 节中，我们使用 `getAttribute` 和 `setAttribute` 来访问元素属性。这些方法将属性值都当成字符串来处理，并不关注它们代表的含义。如果一个元素的宽度为 `width="10em"`，另一个元素的宽度为 `width="100px"`，比较它们的宽度时，就需要将数值从字符串中提取出来，然后找到当前字体大小，进行单位转换，再进行比较。

为了让这类事情更简单一点，SVG 元素对象在对应类型的元素上有一些代表关键特性的属性。比如 `SVGCircleElement` 对象（即 SVG 代码中的 `<circle>`）有 `cx`、`cy`、`r` 等属性，`SVGRectElement`（`<rect>` 标记）有 `x`、`y`、`width`、`height` 等属性。

这些属性并不只是简单地存储数字。在大部分情况下，每个属性都含有两个子属性：`baseVal` 和 `animVal`。其中 `animVal` 对象是只读的，当对象产生动画之后会更新，所以它始终代表属性值的当前显示状态。

除此之外，`baseVal` 和 `animVal` 子属性是包含自身数据的复杂对象，使得在处理以不同单位指定的属性值时更容易。对长度和角度来说，`baseVal.value` 和 `animVal.value` 属性始终存储以用户单位保存的值（角度单位为度），而不管在设定属性时使用的是什么单位。

baseVal 和 animVal 对象还包含不同单位之间进行转换的方法。

示例 14-1 展示了使用不同的方法获取一个有动画的椭圆的 *x* 半径和 *y* 半径。输出见图 14-1（具体的输出值受系统的默认字体大小影响）。

示例 14-1：使用 baseVal 和 animVal 属性

http://oreillymedia.github.io/svg-essentials-examples/ch14/baseval_animval.svg

动画标记：

```
<ellipse id="el" cx="50%" cy="20" rx="40%" ry="1em">
    <animate id="animation" attributeName="rx" to="20%"
             begin="indefinite" dur="2s" fill="freeze"/>
</ellipse>
```

文本输出标记：

```
<text y="3em">
    <tspan x="1em" dy="1.5em">getAttribute("rx"):</tspan>
        <tspan x="50%" id="getRx"/>
    <tspan x="1em" dy="1.5em">getAttribute("ry"):</tspan>
        <tspan x="50%" id="getRy"/>
    <tspan x="1em" dy="1.5em">rx.baseVal.value:</tspan>
        <tspan x="50%" id="rxBase"/>
    <tspan x="1em" dy="1.5em">ry.baseVal.value:</tspan>
        <tspan x="50%" id="ryBase"/>
    <tspan x="1em" dy="1.5em">rx.baseVal.valueAsString:</tspan>
        <tspan x="50%" id="rxBaseString"/>
    <tspan x="1em" dy="1.5em">ry.baseVal.valueInSpecifiedUnits:</tspan>
        <tspan x="50%" id="ryBaseUnits"/>

    <tspan style="font-weight:bold;" x="1em" dy="2.5em">After
        rx.baseVal.convertToSpecifiedUnits():</tspan>
    <tspan x="1em" dy="1.5em">rx.baseVal.valueAsString:</tspan>
        <tspan x="50%" id="rxBaseUnits"/>

    <tspan style="font-weight:bold;" x="1em" dy="2.5em">After
        approx. 1 second of animation:</tspan>
    <tspan x="1em" dy="1.5em">rx.animVal.value:</tspan>
        <tspan x="50%" id="rxAnim"/>
    <tspan x="1em" dy="1.5em">ry.animVal.value:</tspan>
        <tspan x="50%" id="ryAnim"/>
</text>
```

脚本：

```
var doc = document;
var el = doc.getElementById("el"); ❶
doc.getElementById("getRx").textContent = el.getAttribute("rx"); ❷
doc.getElementById("getRy").textContent = el.getAttribute("ry");
doc.getElementById("rxBase").textContent = el.rx.baseVal.value; ❸
doc.getElementById("ryBase").textContent = el.ry.baseVal.value;
```

```
doc.getElementById("rxBaseString").textContent =
 el.rx.baseVal.valueAsString; ❹
doc.getElementById("ryBaseUnits").textContent =
 el.ry.baseVal.valueInSpecifiedUnits; ❺

el.rx.baseVal.convertToSpecifiedUnits(SVGLength.SVG_LENGTHTYPE_EMS); ❻
doc.getElementById("rxBaseUnits").textContent =
  el.rx.baseVal.valueAsString;

var animate = doc.getElementById("animation"); ❼
try {
  animate.beginElement(); //动画开始
} catch(e){/* 如果不支持动画,捕获异常 */}
setTimeout(getAnimatedValue, 1000); ❽

function getAnimatedValue() { ❾
  try {
    animate.endElement(); //动画暂停
  } catch(e){}
  doc.getElementById("rxAnim").textContent = el.rx.animVal.value;
  doc.getElementById("ryAnim").textContent = el.ry.animVal.value;
}
```

❶ el 是指 SVGEllipseElement 对象，也就是椭圆标记。

❷ el.getAttribute("rx") 返回带单位的属性值，也就是标记中写的值。

❸ el.rx.baseVal.value 是 rx 被转换为用户单位之后的值。

❹ el.rx.baseVal.valueAsString 是完整的字符串，包含单位。

❺ el.ry.baseVal.valueInSpecifiedUnits 是数值类型的属性值，但是单位是设定属性时使用的单位。

❻ convertToSpecifiedUnits(unitConstant) 方法可以用来转换单位（本例中转换为了 em）。该方法并不直接返回任何值，而是会改变对象中的 valueAsString 和 valueInSpecifiedUnits 属性值。value 属性的单位始终是用户单位，不受影响。

❼ 该标记定义了一个动画元素，它会改变椭圆的 rx 属性。属性 begin="indefinite" 阻止了动画自动开始，而是通过调用 beginElement() 方法开始动画。在调用时我们将它放在一个 try/catch 块中，以防止在不支持动画的浏览器中出错。

❽ setTimeout() 函数调用告诉浏览器先等待 1 秒（1000 毫秒），然后再执行 getAnimatedValue 函数。

❾ getAnimatedValue() 通过 animate.endElement() 来结束动画，然后查询各个属性的 animVal 属性。尽管 ry 并没有产生动画，但 ry.animVal 仍然存在，它的值和 ry.baseVal 一样。

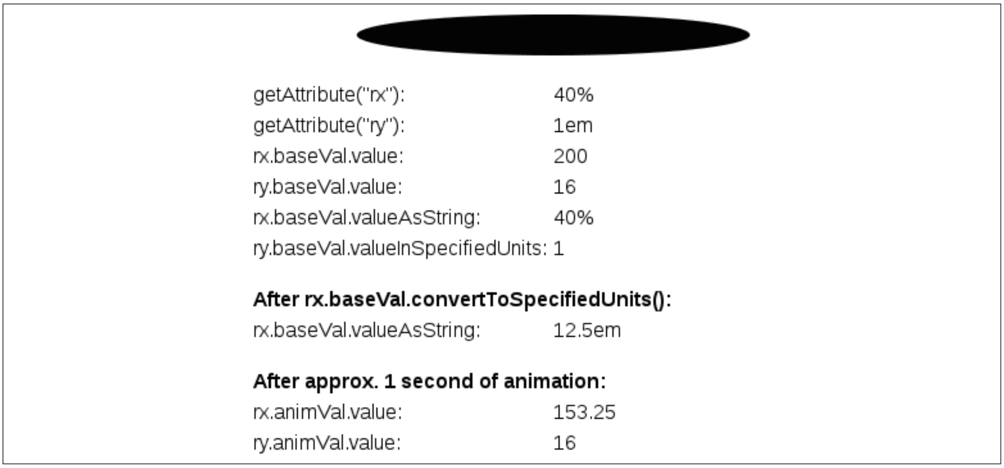


图 14-1：使用 `baseVal` 和 `animVal` 取值的动画

示例 14-1 中使用的 `rx`、`ry` 属性都是 `SVGAnimatedLength` 对象的实例。SVG 定义了一些自定义对象来表示不同的几何数据。表 14-1 列出了一些最重要的对象以及可进行的操作。

表14-1：SVG数据对象

对象名称	描述	属性和方法
`SVGLength`	带单位的长度。使用 `SVGLength.SVG_LENGTHTYPE_unit` 定义，其中 unit 是 `NUMBER`（用户单位）、`PERCENTAGE`、`EMS`（em 单位）、`EXS`（ex 单位）、`PX`、`CM`、`MM`、`IN`、`PT`、`PC` 之一	**属性：** `unitType` 允许的单位之一 `value` 长度（用户单位） `valueInSpecifiedUnits` 长度（`unitType` 单位） `valueAsString` 数值和单位一起的字符串 **方法：** `newValueSpecifiedUnits(unitType, valueInSpecifiedUnits)` 设置值和单位 `convertToSpecifiedUnits(unitType)` 改变单位，并保持值与用户单位下相同
`SVGAngle`	带单位的角度。使用 `SVGAngle.SVG_ANGLETYPE_unit` 定义，其中 unit 是 `UNSPECIFIED`（默认为角度）、`DEG`、`RAD`、`GRAD` 之一	与 `SVGLength` 一样
`SVGRect`	使用用户单位的矩形区域。SVGRect 对象与 `<rect>` 元素（使用 `SVGRectElement` 接口）不是一回事	**属性：** `x`、`y`、`width`、`height` 所有数字都是用户坐标

（续）

对象名称	描述	属性和方法
SVGPoint	在用户空间中的一个点	**属性：** x、y 数字（用户坐标） **方法：** matrixTransform(matrix) 返回通过矩阵变换之后的点的值，参数一个矩阵，为 SVGMatrix 对象
SVGMatrix	用于变换的矩阵，见附录 D	**属性：** a、b、c、d、e、f 数字，以从上到下、从左到右的顺序代表矩阵中的数字 **方法：** multiply(secondMatrix) 返回两个矩阵相乘的结果 inverse() 如果可能的话，转置矩阵，如果不能转置（如 scale(0) 操作），则会抛出异常 translate(x, y)、scale(scaleFactor)、scaleNonUniform(scaleFactorX, scaleFactorY)、rotate(angle)、rotateFromVector(x, y)、flipX()、flipY()、skewX(angle)、skewY(angle) 返回经过指定变换（矩阵相乘）之后的矩阵；rotateFromVector 会计算 x 轴与向量对齐时需要旋转的角度；flipX 和 flipY 等价于在缩放时为 scale 设置 *x*/*y* 值为 −1
SVGTransform	变换命令。使用 SVG_TRANSFORM_type 定义，其中 type 是 MATRIX、TRANSLATE、SCALE、ROTATE、SKEWX、SKEWY 之一	**属性：** type 前述类型之一 matrix SVGMatrix 对象，代表本次变换 angle 使用旋转或斜切时的角度，否则为 0 **方法：** setMatrix(matrix)、setTranslate(x, y)、setScale(scaleFactorX, scaleFactorY)、setRotate(angle, cx, cy)、setSkewX(angle)、setSkewY(angle) 通过指定变换改变 SVGTransform 对象
SVGTransformList	变换（SVGTransform 对象）列表，顺序与指定 transform 属性时相同。除了用于变换的方法外，这些对象还实现了 numberOfItems 属性以及下面通用列表中描述的方法	**方法：** createSVGTransformFromMatrix(matrix) 从 SVGMatrix 创建一个新 SVGTransform 对象 consolidate() 将列表中所有的变换合并为一个变换矩阵，返回 SVGTransform 对象

（续）

对象名称	描述	属性和方法
SVGXxxList	任何可能被当成列表或数组（以及大部分能使用动画）的数据类型，都有对应的 SVGDatatypeList。SVG DOM 的大部分 JavaScript 实现使用数组来代表这些列表，所以才可以使用数组的形式来设置或获取条目	**属性：** numberOfItems 列表长度。 **方法：** clear() 移除列表中的所有条目 initialize(newItem) 清除列表中已有的所有条目，放入新条目 getItem(index) 获取列表中指定索引位置的条目（第一个索引为 0） insertItemBefore(newItem, index)、replaceItem(newItem, index)、removeItem(index)、appendItem(newItem) 顾名思义的一些修改列表的方法
SVGAnimatedXxx	几乎每种数据类型都有 SVGAnimatedDatatype 接口，用于能使用动画的属性值。比如 SVGCircleElement 对象的 cx 属性是一个 SVGAnimatedLength 对象，而 transform 属性则是 SVGAnimatedTransformList 类型	**属性：** baseVal 包含属性值的对象 animVal 只读对象，表示属性在动画过程中的当前展示值

14.2 SVG接口方法

在使用脚本操作 SVG 时，有时候希望能计算一些未并在属性中直接定义的几何属性。比如你可能希望不管文本使用的什么字体，都在文本周围画一个刚好能匹配的矩形。或者希望知道脚本运行时某个动画进行到哪里了。或者希望维护一个复杂属性，比如路径中的曲线部分或变换（transform）属性。

使用 document.getElementById(id) 会返回一个对象，这个对象有很多有用的属性和方法可以进行计算和维护。规范中以接口的概念定义了这些属性和方法，不同类型的对象的属性和方法也各不相同。

表 14-2 列出了本书中描述的一些元素对应的接口的部分特性。这个列表并不全面，但应该足够入门使用了。

表14-2：SVG元素的接口

应用元素	方法或属性	结果
SVGElement（SVG 命名空间中的任何元素）	.ownerSVGElement()	返回最近的祖先 <svg> 元素，如果是在顶级 SVG 元素调用，则返回 null
	.viewportElement()	返回建立当前元素 viewport 的 <svg>、<pattern>、<symbol> 或 <marker> 元素

（续）

应用元素	方法或属性	结果
SVGLocatable（占据坐标空间的元素或者含有变换属性的元素：图形元素、<g> 或者 <svg>）	.nearestViewportElement	建立当前元素 viewport 的 <svg>、<pattern>、<symbol> 或 <marker> 元素
	.farthestViewportElement	包含当前元素的最顶层 <svg> 元素
	.getBBox()	以 SVGRect 对象的形式返回当前元素的位置和边界。包含 x、y、width、height 属性，分别代表坐标和能包含整个图形的最小矩形框。边界不受笔画宽度、裁剪、蒙版、滤镜影响
	.getCTM()	返回一个 SVGMatrix 对象，代表从当前元素的坐标系统到 nearestViewportElement 坐标系统之间的变换
	.getScreenCTM()	返回一个 SVGMatrix 对象，代表从当前元素的坐标系统到文档最顶层的屏幕坐标或者客户端坐标之间的变换矩阵。这个方法在处理事件时可以很方便地转换鼠标 / 指针使用的坐标和图形坐标
	.getTransformToElement(*SVGElement*)	返回一个 SVGMatrix 对象，代表当前元素坐标到指定元素坐标之间的变换
SVGTransformable（能使用 transform 的任何元素）	.transform	代表定义在元素上的变换原始值和动画过程中的值的 SVGAnimatedTransformList
SVGStylable（能使用 style 属性的任何元素）	.style	返回一个 CSSStyleDeclaration 元素，代表内联在元素上的样式。style 对象的方法可以参见 13.4 节
SVGSVGElement（<svg>）	.suspendRedraw(*maxWaitTimeInMilliseconds*)	告诉浏览器在绘制图形的时候暂停指定时间（最长一分钟）。如果你希望做很多修改，但是在最后一起应用，则这种方法很有用。该方法返回一个数字 ID，稍后可以将它传给 unsuspendRedraw
	.unsuspendRedraw(*suspendID*)	取消 ID 代表的由 suspendRedraw 创建的暂停
	.unsuspendRedrawAll()	取消 SVG 上所有由 suspendRedraw 创建的暂停，恢复 SVG 绘制
	.pauseAnimations()	暂停 SVG 上所有 SMIL 动画的时钟
	.unpauseAnimations()	恢复 SVG 上所有 SMIL 动画的时钟
	.animationsPaused()	返回 true 或 false，代表动画是否由上方的方法暂停
	.getCurrentTime()	返回 SMIL 动画使用的时钟的当前值。正常情况下，这个值为文档加载之后的秒数，但是暂停动画或者是直接修改这个值都可能影响
	.setCurrentTime(*timeInSeconds*)	修改 SMIL 时钟的值，影响所有的动画

（续）

应用元素	方法或属性	结果
SVGSVGElement (<svg>)	.getIntersectionList(*rectangle*, *referenceElement*)	返回一个元素列表，这些元素均在`` 元素坐标系统下与指定的矩形（SVGRect 对象）有重叠，并且这些元素全部是指定元素的子元素。只有能响应指针事件（取决于 pointer-events 属性，默认可以响应）的元素才被返回。指定的元素可以为 null，表示只要是 <svg> 的子元素都可以 [a]
	.getEnclosureList(*rectangle*, *referenceElement*)	与 getIntersectionList() 类似，区别在于只返回完全在矩形区域中的元素
	.checkIntersection(*element*, *rectangle*)	返回 true 或 false，代表在 <svg> 元素所在坐标系下指定元素与指定 SVGRect 对象是否重叠
	.checkEnclosure(*element*, *rectangle*)	返回 true 或 false，代表在 <svg> 元素所在坐标系下指定元素是否被包含在指定 SVGRect 对象中
	.createSVG*Xxx*()	SVGSVGElement 支持一些方法用来创建表 14-1 中的各种数据对象（SVGPoint、SVGAngle、SVGMatrix 等）。这些方法不接受参数，结果会被初始化为 0（除了 createSVGMatrix() 会返回一个矩阵）
SVGUseElement (<use>)	.instanceRoot	包含 <use> 元素代表的图形 shadow DOM 树的最顶层节点。shadow DOM（SVGElementInstance 对象）中的元素只有有限的方法：你不能直接操作属性或者样式，但是它们可能是用户事件的目标元素。每个 SVGElementInstance 都有一个 correspondingElement 属性，链接到它复制出来的图形元素，还有一个 correspondingUseElement 属性链接回 <use> 元素 [b]
SVGPathElement (<path>)	.getTotalLength()	以用户坐标返回计算后整个路径的长度（不计 move 命令）。这个值在不同的客户端中不一定完全一样，因为在计算一些曲线的时候会有舍入
	.getPointAtLength(distance)	返回以用户单位计算的距离起点 distance 单位的点，为一个包含 x 和 y 属性的 SVGPoint 对象。计算方法和 getTotalLength() 一样
SVGPathData (<path> 元素和其他支持路径数据属性的元素，如 <animateMotion>)	.pathSegList	包含一个代表路径各个部分的对象列表。这个列表可以使用面向对象的方式被查询或修改，具体方法请参见 SVG 规范（）。pathSegList 属性返回与 d 属性对应的路径。如果 d 属性有动画的话，则需要使用 animatedPathSegList 属性获取路径当前状态的对象列表

（续）

应用元素	方法或属性	结果
SVGPathData（<path>元素和其他支持路径数据属性的元素，如 <animateMotion>）	.normalizedPathSeglist	简化版的路径各部分列表，每个部分被转化为绝对坐标下的移动、画线、曲线或者关闭路径等命令。与上面类似，这个属性也有一个对应的的 animatedNormalizedPathSegList 属性
SVGAnimatedPoints（<polygon> 和 <polyline> 元素）	.points	返回一个 SVGPointList，代表与元素相关的点的列表。animatedPoints 属性与之类似，代表有动画的属性
SVGTextContentElement（任何文本元素，包括 <text>、<tspan> 和 <textPath>）	.getNumberOfChars()	返回元素中字符的总数，包括所有的 <tspan> 元素。多字节字符会用 UTF-16 来呈现，然后计数
	.getComputedTextLength()	返回在用户坐标系下应用了所有的 CSS 属性和 dx、dy 属性之后包含整个文本的长度。不包括在 textLength 属性基础上做的调整
	.getSubStringLength(*charNum, nChars*)	返回截取的字符串一部分的长度。截取的方法是从索引为 charNum 开始（第一个字符索引是 0），截取 nChars 个字符，如果不足的话就到字符串结束为止
	getStartPositionOfChar(*charNum*)	返回一个带 x 和 y 属性的 SVGPoint 对象，表示在用户坐标系中指定字符的起始位置。字符与起始位置的相对关系取决于书写模式（横向或纵向、从左到右或从右到左）以及 baseline-alignment 属性。默认情况下从左往右的文本中，起始点位于字符的最左侧基线位置
	.getEndPositionOfChar(*charNum*)	与 getStartPositionOfChar 类似，但返回的位置代表字符结束时所在的基线位置
	.getExtentOfChar(*charNum*)	与 getBBox() 类似，返回一个 SVGRect 对象，区别在于该方法针对单个字符
	.getRotationOfChar(*charNum*)	返回指定字符在经过所有变换之后的旋转角度（以度为单位），不包括坐标系的变换
	.getCharNumAtPosition(*point*)	返回指定位置的字符索引值，如果指定的点没有字符的话返回 −1。位置使用 SVGPoint 对象指定，可以是从某些接口方法返回的，也可以在 <svg> 元素上调用 createSVGPoint() 然后设置 x 和 y 属性的

（续）

应用元素	方法或属性	结果
ElementTimeControl 和 SVGAnimationElement（任何 SVG 动画元素：<animate>、<set>、<animateTransform> 和 <animateMotion>）	.targetElement	SVG 动画修改的 SVGElement 元素
	.beginElement()	如果动画没有被 restart 属性 never 或者 whenNotActive 阻止的话，立即开始动画
	.beginElementAt(*offset*)	在 offset 秒之后开始动画。如果 offset 是负值，则动画会立即开始，但进程会像已经开始了 -offset 秒之后一样
	.endElement()	立即结束当前正在运行的动画（包括重复的）
	.endElementAt(*offset*)	offset 秒后结束当前动画
	.getStartTime()	如果动画正在运行，返回动画相对 SVG 时钟的开始时间。如果动画还未开始，则返回动画将要开始的时间。其他情况下，抛出错误
	.getCurrentTime()	返回 SVG 动画时钟的当前时间（单位为秒），与在 SVG 元素上调用 getCurrentTime() 效果一样
	.getSimpleDuration()	返回动画每一轮的时长（dur 属性），如果未定义则抛出错误

[a] 截稿时，Firefox 30 不支持 getIntersectionList、getEnclosureList、checkIntersection、checkEnclosure。

[b] Firefox 30 也不支持 <use> 元素的 shadow DOM 树访问，也未实现 SVGElementInstance 接口。

[c] IE11 不支持 SMIL 动画，因此动画相关的属性和方法在其中都未实现。Apache Batik SVG viewer 1.7 中，使用 beginElement 和 beginElementAt 会抛出错误。

14.3 使用ECMAScript/JavaScript创建SVG

下面这个例子是一个简单的模拟时钟，如图 14-2 所示。示例 14-2 是 SVG 代码。

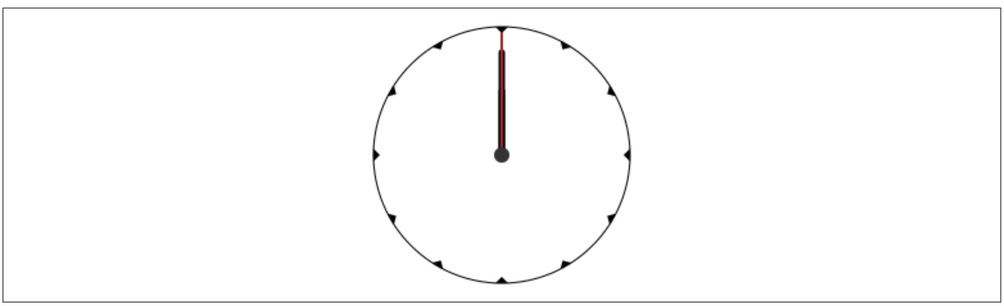


图 14-2：模拟时钟

示例 14-2：模拟时钟的 SVG 代码

```
<svg xmlns="http://www.w3.org/2000/svg"
    id="clock" width="250" height="250" viewBox="0 0 250 250">
<title>SVG Analog Clock</title>
```

```
<circle id="face" cx="125" cy="125" r="100"
        style="fill: white; stroke: black"/>
<g id="ticks" transform="translate(125,125)">
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(30)"  />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(60)"  />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(90)"  />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(120)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(150)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(180)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(210)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(240)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(270)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(300)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(330)" />
    <path d="M95,0 L100,-5 L100,5 Z" transform="rotate(360)" />
</g>

<g id="hands" style="stroke: black;
                     stroke-width: 5px;
                     stroke-linecap: round;">
  <path id="hour" d="M125,125 L125,75"
        transform="rotate(0, 125, 125)"/>
  <path id="minute" d="M125,125 L125,45"
        transform="rotate(0, 125, 125)"/>
  <path id="second" d="M125,125 L125,30"
        transform="rotate(0, 125, 125)"
        style="stroke: red; stroke-width: 2px" />
</g>
<circle id="knob" r="6" cx="125" cy="125" style="fill: #333;"/>
</svg>
```

这段代码不是特别复杂，却显得很冗余。标识 12 个小时的 12 个元素的写法几乎完全一样，只有 rotation 属性不同。你可以使用 <use> 元素来避免重复路径数据，但它也不会简化多少工作。如果你希望给它们加上数字，还需要 12 个 <text> 元素。而如果你希望添加分钟标记，则需要添加 60 个元素。

在写程序时，如果需要重复做某件事情很多次，你会很自然地使用循环或者函数。在 13.4.6 节，我们使用了 JavaScript 来基于用户输入创建任意数量的 SVG 元素。在本例中，我们也可以使用同样的方法来创建图形，尤其是在有很多重复图形且是比较规则的几何图形时更应该如此。

示例 14-3 在支持 JavaScript 的 SVG 阅读器中创建了和之前一样的输出。它使用了基本的 DOM 方法和本章介绍过的 SVG DOM 特性。<svg> 元素仅包含两个标记元素：一个 <title> 和一个 <script>。

示例 14-3：使用 ECMAScript 创建模拟时钟

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
```

```
    id="clock" width="250" height="250" viewBox="0 0 250 250"
     onload="init()" >
<title>Scripted Analog Clock</title>

<script type="application/ecmascript"> <![CDATA[

  /* 图形会包含在<svg>对象中 */
  var clock; ❶

  function init() { ❷
    /* 选择空<svg>元素 */
    clock = document.getElementById("clock");
    var svgns = clock.namespaceURI,
        doc = document;

    clock.suspendRedraw(1000); ❸

    /* 创建表盘 */ ❹
    var face = doc.createElementNS(svgns, "circle");
    face.cx.baseVal.value = 125;
    face.cy.baseVal.value = 125;
    face.r.baseVal.value = 100;
    face.style.cssText = "fill: white; stroke: black";
    clock.appendChild( face );

    /* 创建小时标识的组 */
    var ticks = clock.appendChild(
          doc.createElementNS(svgns, "g") );
    ticks.setAttribute("transform", "translate(125,125)" );

    /* 创建小时标识 */
    var tickMark;
    for (var i = 1; i <= 12; i++) { ❺
      tickMark = doc.createElementNS(svgns, "path");
      tickMark.setAttribute( "d",
                          "M95,0 L100,-5 L100,5 Z" );
      tickMark.setAttribute( "transform",
                          "rotate(" + (30*i) + ")" );
      ticks.appendChild( tickMark );
    }

    /* 创建时针 */
    var hands = clock.appendChild(
                    doc.createElementNS(svgns, "g") );
    hands.style.cssText =
      "stroke: black; stroke-width:5px; stroke-linecap: round;";

    var hourHand = hands.appendChild(
                    doc.createElementNS(svgns, "path") );
    hourHand.id = "hour";
    hourHand.setAttribute("d", "M125,125 L125,75"); ❻
    hourHand.setAttribute("transform", "rotate(0, 125, 125)");

    /* 省略类似的创建分针和秒针的代码 */
```

```
        /* 创建中心的元素 */ ❼
        var knob = doc.createElementNS(svgns, "circle");
        knob.setAttribute("cx", "125");
        knob.setAttribute("cy", "125");
        knob.setAttribute("r", "6");
        knob.style.setProperty("fill", "#333", null);
        clock.appendChild( knob );

        clock.unsuspendRedrawAll(); ❽
      }

    // ]]>
  </script>
</svg>
```

❶ 脚本声明了一个全局变量来保存 `<svg>` 元素，但大量的代码在初始化函数（`init()`）中，这个函数在 SVG 加载后才会运行。

❷ `init()` 函数开头选择了 `<svg>` 元素，同时还声明了一些变量以便后续使用。当创建一个新元素时，从任何一个已有的元素上取 `.namespaceURI` 属性是一个好方法，这样可以避免手工输入一大串 URL。

❸ 尽管不是必须的，但是在进行大量 DOM 操作之前暂停 SVG 更新绘制可以提升在某些阅读器中的性能。

❹ 创建了一个 `<circle>` 元素作为表盘，样式和属性使用 SVG DOM 属性来指定，然后将这个元素添加到 SVG 中。

❺ 这里就是循环的过程。`for` 循环运行其中的代码 12 次来创建 12 个小时的标记，并计算它们旋转的角度。值得注意的是，对初始化复杂的属性（比如 `d` 或者 `transform`）来说，使用 `setAttribute()` 传入一个字符串会比维护复杂的 DOM 对象更容易。

❻ 指针初始化时指向 0 点，旋转角度为 0，这样后续变换角度时计算更容易。

❼ 为了与表盘对比，中心的圆 `<circle>` 使用设置 DOM 属性的方式来初始化。这还是需要 6 行代码。

❽ DOM 构建完毕时记得取消之前的 `suspendRedraw()` 调用！

使用脚本简化了绘制小时标记的代码，但是增加了绘制简单元素（比如表盘的圆）的代码量。这也是像 Snap.svg 这样的 JavaScript 库得以流行的主要原因，它们对一些常用操作有更简单的方法，比如创建元素或者设置属性之类。

我们会在稍后介绍这部分内容，但在开始讨论库（更简单的方法）之前，还是继续使用原始的 SVG 来绘制表盘。毕竟，我们还面临着一个大问题：我们的钟无法告诉我们当前时间。

14.4　使用脚本控制动画

让时钟走起来的一种方法是在指针上使用 `<animateTransform>` 元素。你可以让秒针每

分钟旋转 360 度，分针每小时旋转 360 度，时针每 12 小时旋转 360 度。但即便如此，`<animateTransform>` 仍然不能让时钟显示正确的时间。[1]

除此之外，还有一些仅使用动画元素不能轻易完成的事情。比如它们不能响应过去的用户事件，也不能对未来的事件作出有针对性的响应。如果动画依赖于逻辑、数据或者复杂的用户交互，那么使用（其他程序中的）脚本来控制动画就比直接定义在 XML（文档结构）中更合理。

使用 JavaScript 创建动画的方法就是在动画过程中不断修改需要动画的属性，直到从初始值变为结束时的值。如果你刚刚开始学习编程的话，可能会想到使用 `while(true)` 循环来持续更新属性。

这的确能保证你的时钟始终是准确的，但不推荐这样做。使用 `while(true)` 的话相当于在频繁地询问时钟："现在什么时间了？"这样会使用程序没有空闲时间来做其他的事情。在 SVG 脚本中，则会让计算机没有机会处理其他的任务。

其实在上述例子中，每一次循环只需要不到 1 毫秒的时间，让指针为这么短的时间产生旋转完全没有意义。相较而言，大部分的电影、视频会每秒更新画面 30 到 60 次。这个速度被称为视频的帧率（frame rate）[2]，30~60 次已经足够让人眼认为画面是连续的了。

计算机的显示也有帧率。当内容变化时，显示器上画面的一部分或者整个画面会更新。但是，计算机显示的帧率取决于计算机在后台有多少工作量。如果计算机陷入无休止的循环中，则没有时间来重绘屏幕，这样不管你更新属性有多快，你的动画都会变得很慢或者很卡。

为了让计算机创建出流畅的动画，你需要更礼貌一些。当调用 `requestAnimationFrame(animationfunction)` 时，你其实在说："计算机，下次当你准备重绘屏幕的时候，请先运行这个函数。"函数被调用时会传入一个时间戳，这个时间戳在同一帧的函数中都是一样的值，你可以使用这个值来进行动画运算。（这个时间戳基于文档时钟，而不是系统时钟，因此不能用来设置时间。）

如果你在动画函数的最后使用 `requestAnimationFrame` 并将它自己作为参数传进去，则计算机会以尽量高的频率在屏幕上绘制动画，这样可以创建一个很流畅的动画而又不会消耗太多的计算机资源。[3] 需要注意，如果包含脚本的容器被最小化或者被隐藏，则在它显示出来之前，动画函数都不会被调用。

注 1：SMIL 规范定义了与系统时钟同步动画起始时间的格式，但在大部分的浏览器和 SVG 浏览器中都未实现。

注 2：也可简写为 FPS，frames per second。——译者注

注 3：你可能听过一个编程术语叫递归，即一个函数调用它自己。这里并不是递归，因为你的函数调用的是 `requestAnimationFrame()`，而不是它们自己。

使用 setTimeout 模拟 requestAnimationFrame

相对来说，requestAnimationFrame() 在 DOM 规范中还是一个比较新的方法，有一些比较老的浏览器以及 Batik 软件并不支持。为了能让脚本顺利执行，可以先测试这个函数是否存在，如果不存在的话先像下方代码那样模拟一个。这个模拟的方法使用了 setTimeout(function, waitTime) 方法，它可以告诉计算机在 waitTime 毫秒之后再运行函数。下面的代码应该放在 <script> 最前面：

```
if (!window.requestAnimationFrame) { ❶

  window.requestAnimationFrame = function(animationFunction) { ❷

    function wrapperFunction() { ❸
      animationFunction(Date.now());
    }

    setTimeout(wrapperFunction, 30); ❹
  }
}
```

❶ 如果 requestAnimationFrame 方法不存在……

❷ 创建你自己的函数，并将它保存到全局对象的 requestAnimationFrame 属性中。这个函数必须接受一个动画回调函数作为参数。

❸ 将传入的动画函数包裹在一个不接受参数的函数中。包裹函数调用动画函数时会传入时间戳作为参数。Date.now() 返回的是整型的系统时间戳。

❹ setTimeout 方法会在计算机空闲的时候调用动画函数，但是调用间隔不会小于 30 毫秒，即大约每秒 33 帧。

setTimeout() 函数考虑的情况没有 requestAnimationFrame() 那么细致，它不会调整计算机当前正在做的事情，而且无论窗口是否可见都会始终执行动画函数。这段示例代码也不能完全替代 requestAnimationFrame()，尤其是它不会将多个动画调用与 SMIL 动画时钟“对齐”，也不能取消某个动画帧的调用请求。不过，由于我们使用的调用等待时间还算比较长，对示例程序来说，动画的流畅度应该还是能接受的。使用时将它插到你的代码最前面（或者在文件最前面放入单独的 script 标记）。

示例 14-4 展示了使用 requestAnimationFrame() 更新时钟指针的代码。

示例 14-4：脚本驱动的 SVG 时针动画

全局变量：

```
/* 引用代表指针的路径元素 */
var hourHand,
    minuteHand,
    secondHand;
/* 引用旋转指针的SVGTransform对象 */
```

```
var secondTransform,
    minuteTransform,
    hourTransform;
/* 时间常量 */
var secPerMinute  = 60,
    secPerHour    = 60*60,
    secPer12Hours = 60*60*12;
```

初始化变量（**init()** 函数）：

```
function init() {
  /*
    获取引用代表指针的路径元素
  */
  hourHand = document.getElementById("hour");
  minuteHand = document.getElementById("minute");
  secondHand = document.getElementById("second");

  /* 获取代表当前旋转的变量矩阵SVGTransform对象rotate(0, 125, 125)：
  */
  secondTransform = secondHand.transform.baseVal.getItem(0);
  minuteTransform = minuteHand.transform.baseVal.getItem(0);
  hourTransform = hourHand.transform.baseVal.getItem(0);
  updateClock(); /* 让时钟开始运行 */
}
```

updateClock 函数：

```
function updateClock() {
    /* 获取系统时间 */ ❶
    var date = new Date();
    /* 计算从0点开始过去的秒数 */
    var time = date.getMilliseconds()/1000 +
               date.getSeconds() +
               date.getMinutes()*60 +
               date.getHours()*60*60; ❷

    /* 计算旋转角度 */ ❸
    var s = 360*( time % secPerMinute )/secPerMinute,
        m = 360*( time % secPerHour )/secPerHour,
        h = 360*( time % secPer12Hours )/secPer12Hours;

    /* 使用SVGTransform.setRotate(angle, cx, cy)来更新旋转角度：*/
    secondTransform.setRotate( s, 125, 125); ❹)
    minuteTransform.setRotate( m, 125, 125);
    hourTransform.setRotate( h, 125, 125);

    window.requestAnimationFrame( updateClock ); ❺
    // 重复下一帧
}
```

❶ 构造函数 `new Date()` 以 JavaScript 对象的形式返回系统日期和时间（精确到千分之一秒）。

❷ `date.getPart()` 形式的函数会以整形返回日期的一部分。使用这些方法，可以将当前时间计算成与今天 0 点的差值（以秒为单位）。

❸ 每个指针的旋转角度（自零点开始旋转的角度）使用之前定义的常量进行计算。`%` 是取模运算符，返回整除后的余数。

❹ 在初始化函数中，我们访问了每个指针 `transform` 属性中 `SVGTransformList` 中的第一个（也是唯一一个）变换的 `baseVal`，并将这个 `SVGTransform` 对象存到变量中。每次更新的时候，我们直接修改这个对象的旋转角度。直接修改这个对象而不是使用 `setAttribute`，这可以让浏览器省去解析属性字符串的时间。

❺ `updateClock` 的最后一行使用 `requestAnimationFrame` 来让自己在下次屏幕刷新时再次被调用。

动画循环是从初始化函数的最后一行 `updateClock()` 调用开始的。初始化函数本身则是通过在 `<svg>` 开始标记中添加 `onload="init()"` 来运行。最后，一个完整的时钟可以在本书的网站上看到：http://oreillymedia.github.io/svg-essentials-examples/ch14/animated_clock_js.svg。

14.5 使用JavaScript库

在示例 14-3 中，我们通用手工调用 JavaScript 创建元素的方式创建了整个时钟。在这个过程中能感觉到，使用这种方式生成元素是痛苦而效率低下的。应该有一种更好的方式，事实上，也真的有。你可以使用免费的开源 JavaScript 库，比如 D3.js（http://d3js.org/）、Raphaël（http://raphaeljs.com/）、Snap.svg（http://snapsvg.io/）等来简化工作。这些库都属于外部脚本，暂时只考虑了 Web 浏览器环境。不过，它们都有一系列很好用的方法，你自己编写的时候可以借用。

使用什么库取决于需求。D3.js 是一个“维护基于数据产生的文档的 JavaScript 库”。它最适用于维护一系列相似元素，定义它们的属性、样式以及根据数据数组中对应的值来响应事件。例如，如果你需要一个可互动性很强的柱状图表，那么 D3.js 会是一个很好的选择。

Raphaël 和 Snap.svg 则是更通用的库，它们的目标是使得创建和修改带动画的图形更加容易。Raphaël 还通过将图形转换为其他形式的图形命令的方式与老版本浏览器兼容。Snap 则是由 Adobe Web Platform 团队推出的，它为现代浏览器而生，只使用了 SVG。下面的例子将使用 Snap。

这些库都会将 DOM 元素包裹到自定义对象中，这个对象有一些额外的属性和方法可用。在 Snap 中，包含图形的 `<svg>` 元素被包裹为 `Paper` 对象。Snap 提供了一些方法可供调用，可以用这些方法来添加图形元素、修改属性和处理事件。

用于增强 HTML 处理的 JavaScript 库有很多，但在处理 SVG 时请小心选用。这是因为它们没有任何针对图形的方法，甚至如果没有处理好 XML 命名空间的话，在处理普通任务时也可能会遇到问题。比如，流行的 jQuery 库（至截稿时）没有任何在 SVG 命名空间中创建元素的方法，如果你让 jQuery 为你创建一个圆，它会返回一个 `HTMLUnknownElement` 对象。而针对 SVG 写的库，比如 D3 和 Snap，知道 SVG 中的元素名，也就知道 `circle` 是表示 `SVGCircleElement`。

下面是基于 Snap 的动画时针的 XML：

```
<!DOCTYPE svg PUBLIC "-//W3C//DTD SVG1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">
<svg xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink"
    id="clock" width="250" height="250" viewBox="0 0 250 250"
    onload="init()" >
  <title>Snap.svg Analog Clock</title>

  <script type="application/ecmascript"
          xlink:href="snap.svg-min.js"></script>
  <script type="application/ecmascript">

  /* 初始化和更新时针的函数放这里 */

  </script>
</svg>
```

第一个 `<script>` 元素引入了 Snap 库。我们直接使用了部署在自己服务器上的压缩版本的脚本，即库的代码已经被去掉了空格和注释，并且变量名也被修改得更短。压缩版本的 Snap.svg v0.3.0 大小是 72 KB。也就是说，下载 SVG 库消耗的流量大约和下载一个中等复杂度的 PNG 图片差不多。

在 HTML 文件中引入外部库需要使用 `src` 属性指定文件 URL，同时要注意必须包含开始和结束标记，空 `<script>` 元素在老版本的浏览器中不能正常工作[4]。

第二段脚本与前面的例子差不多，调用 `init()` 函数绘制时钟，然后使用 `updateClock()` 让它走起来。

如果你自己实验这些例子，请确保使用的是最新的 snap.svg-min.js。这个库最早是为了修改在 HTML 页面中内嵌的 SVG 代码而设计的，早于 0.3.0 的版本运行在独立的 SVG 文件中时有 bug（https://github.com/adobe-webplatform/Snap.svg/issues/88）。

注 4：指不能使用的方式引入脚本。——译者注

示例 14-5 展示了使用 Snap 绘制时钟的代码。它和示例 14-3 的结构（和结果）是一样的，但展示了很多 Snap 方法的使用。

示例 14-5：使用 Snap.svg 绘制模拟时钟

```
/* 绘制时钟的Paper对象 */
var clock;
/* 指向代表指针的Snap元素 */
var hourHand,
    minuteHand,
    secondHand;  ❶

/* 时间转换常量 */
var secPerMinute  = 60,
    secPerHour    = 60*60,
    secPer12Hours = 60*60*12;

function init() { ❷
  /* 选择空的<svg>元素作为Paper对象 */
  clock = Snap("#clock");

  /* 创建表盘 */ ❸
  var face = clock.circle(125, 125, 100);
  face.attr({fill: "white", stroke: "black"});

  /* 创建小时标记 */
  var ticks = clock.g();
  ticks.transform("t125,125"); ❹

  var tickMark;
  for (var i = 1; i <= 12; i++) { ❺
    tickMark = clock.path("M95,0 L100,-5 L100,5 Z");
    tickMark.transform("rotate("+ (30*i) + ")");
    ticks.add(tickMark);
  }

  /* 创建指针 */
  hourHand = clock.path("M125,125 L125,75");
  minuteHand = clock.path("M125,125 L125,45");
  secondHand = clock.path("M125,125 L125,30");

  var hands = clock.g(hourHand, minuteHand, secondHand);  ❻
  hands.attr({stroke: "black",
              "stroke-width": 5, ❼
              "stroke-linecap": "round"});
  secondHand.attr({stroke: "red", strokeWidth: "2px"});

  /* 中间的圆 */
  clock.circle(125, 125, 6).attr({fill: "#333"}); ❽

  updateClock(); ❾
}

function updateClock()
```

```
    {
      /* 调整指针 */
    }
```

❶ 尽管全局变量名是一样的，但它们的内容是不同的。clock 是 Snap 中的 Paper 对象，代表指针的变量也会指向 Snap 包裹后的元素对象。

❷ 在 init() 函数中，Snap(selector) 方法使用一个查询字符串（CSS 选择器格式）创建了一个包含 <svg> 的 Paper 对象。这个函数还可以使用 Snap(width, height) 的方式来调用，将以指定的尺寸创建一个新的 SVG 元素。

❸ Paper.circle(cx, cy, r) 以指定的坐标和半径创建了一个 <circle>，并将它加入 SVG 中。这个 Snap 包裹后的对象被赋值给一个变量，以便在下一行修改它的属性。Element.attr(attrValues) 函数可以传入一个 JavaScript 对象一次设置多个属性。（我们使用表示属性是因为截止到 v0.3.0，Snap 都没有提供直接设置内联样式的方法。）

❹ 使用 Paper.g() 创建了一个空的 <g> 元素 ticks。可以使用 ticks.attr() 来设置 transform 属性，也可以使用 Snap 为这种常见任务提供的快捷方法。变换的命令 translate(125,125)t125,125 也有简便的写法，可以写成 t125,125。

❺ for 循环中使用 Paper.path(pathData) 创建了时针标识，稍后会使用 ticks.add(tickMark) 将这些标识移到 <g> 中。这里使用了标准 SVG 语法设置变换属性，只是为了展示两种方法都是可行的。

❻ 时钟指针被创建后，接下来使用 Paper.g(Element, Element, ...) 创建一个组，并将这些元素移到组中。

❼ 当使用 JavaScript 对象语法设置属性的时候，如果属性名中含有中杠（-），你需要将它用引号包裹起来（“stroke-width”）或者将它转成驼峰式（strokeWidth）。

❽ 因为 Snap 在创建元素的时候也会返回这些元素，所以你可以将方法连缀起来。

❾ 和前面的例子一样，初始化函数的最后一行是调用 updateClock()。

如果你在浏览器中载入这个文件，会发现它和图 14-2 完全一样。上面的 SVG 做的事情和示例 14-3 一样，但是代码更好懂、更炫酷了。

剩下的任务就是让我们用 Snap 写的时钟动起来了。代码见示例 14-6。这次我们不再使用 requestAnimationFrame 手工调用动画，而是使用 Snap 的 Element.animate() 方法来处理动画。它会自动调用 requestAnimationFrame 方法，如果浏览器不支持的话，会使用自己定义的模拟方法。

Element.animate() 方法有两个必填参数和两个可选参数。

- 一个属性对象（和 Element.attr() 格式一样），指定动画结束时各个属性的值。
- 以毫秒指定动画时长，即动画属性多久之后达到终值。
- 可选参数，一个用于定义在指定时间内属性变化率的函数（缓动函数）。Snap 源码中定义了一些函数，它们可以通过 Snap 的 mina 对象访问：mina.easeinout 代表平滑的加速

和减速，而 mina.bounce 会以很快的速度到达终值，然后回弹几次，最后才稳定下来。匀速运动请使用 mina.linear。

- 可选函数，一个在动画结束之后会被调用的函数。可以用来产生无限循环的动画。

示例 14-6：使用 Snap.svg 使时钟产生动画

```
function updateClock()
{
    /* 获取系统时间 */ ❶
    var date = new Date();
    /* 计算从0点开始过去的秒数 */
    var time = date.getMilliseconds()/1000 +
               date.getSeconds() +
               date.getMinutes()*60 +
               date.getHours()*60*60;

    /* 计算旋转角度 */
    var s = 360*( time % secPerMinute)/secPerMinute,
        m = 360*( time % secPerHour )/secPerHour,
        h = 360*( time % secPer12Hours )/secPer12Hours;

    secondHand.transform("r" + s +",125,125"); ❷
    minuteHand.transform("r" + m +",125,125");
    hourHand.transform("r" + h +",125,125");

   secondHand.animate({transform: "r" + [s + 360, 125, 125]}, ❸
                      60000, mina.linear);
   minuteHand.animate({transform: "r" + [m + 6, 125, 125]},
                      60000, mina.linear);
   hourHand.animate({transform: "r" + [h + 0.5, 125, 125]},
                      60000, mina.linear, updateClock); ❹
}
```

❶ 计算方法与示例 14-4 一样。

❷ 使用 Snap 的快捷方法和变换的语法来设置时间。

❸ 调用 animate 函数使时钟走起来。每次调用 animate 方法时的第一个参数给出 1 分钟后指针的位置，秒针旋转 360 度，分钟旋转 6 度，时钟旋转 0.5 度。第二个参数表示动画时长为 60 000 毫秒（1 分钟），第三个参数表示动画为匀速（linear）运动。

❹ 1 分钟过后，最后一个动画中指定的回调参数 updateClock 会被调用，使动画再次开始。这会使得时钟再次与系统时钟同步，并开始下一分钟的动画。注意三个动画中只需要一个指定回调函数即可，因为调用 updateClock 会使三个动画都开始运动。

你可以在线观看真实的时钟：

http://oreillymedia.github.io/svg-essentials-examples/ch14/snap_animated_clock.svg

14.6 Snap中的事件处理

使用像 Snap 这样的库也可以使得事件处理更容易。下面的例子很简单，它在 HTML 页面中使用 Snap，在页面上显示一个圆和一个按钮。你可以拖动圆，点击按钮会使圆回到中间。

你需要使用的 Snap 函数主要是 `Snap()`、`click()` 和 `drag()`。`Snap()` 函数接受一个 `#idName` 字符串，并返回代表对应元素的包裹对象。一旦你拿到这个对象，就可以使用 `click()` 和 `drag()` 方法来让元素响应对应的事件。

示例 14-7 展示了需要的 HTML。

示例 14-7：Snap 事件示例的 HTML

```
<html xml:lang="en" lang="en"
    xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Click and Drag Events in Snap</title>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <script type="text/javascript" src="snap.svg-min.js"></script>
  <script type="text/javascript">
    function init() {
    }
  </script>
</head>

<body onload="init()">
  <h1>Click and Drag Events in Snap</h1>

<div style="text-align:center">
  <svg width="200" height="200" viewBox="0 0 200 200"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">

    <circle id="circle" cx="100" cy="100" r="30"
      style="fill:#663399; stroke: black"/>

    <rect id="button" x="60" y="170"
        rx="5" ry="5" width="80" height="25"
        style="stroke:black; fill:#ddd; cursor:pointer"/>
    <text id="buttonText" x="100" y="187" class="buttonText"
      style="fill:black; stroke:none;
      font-family: sans-serif; font-size: 12pt;
      text-anchor:middle; cursor:pointer">Reset</text>
  </svg>
  </div>
</body>
</html>
```

14.6.1 点击对象

要设置一个点击处理方法的话，需要调用 Snap 对象的 `click()` 方法，并传入用于处理点击事件的函数名。下面是用于处理按钮以及按钮内文本的点击的代码：

```
function init() {
  Snap("#button").click(resetFcn);
  Snap("#buttonText").click(resetFcn);
}

function resetFcn(evt) {
  Snap("#circle").attr({cx: 100, cy: 100});
}
```

处理函数的参数是触发事件，但在这个例子中，`resetFcn()` 不需要使用它。到目前为止，如果你使用这些代码，没有任何效果，因为圆已经在中间了。你可以试着改变 `cx` 或者 `cy` 的值，看看处理函数是否正确工作了。

14.6.2 拖拽对象

按钮已经处理完了，现在我们为圆添加拖拽处理。`drag()` 方法接受三个参数：用于处理移动事件的函数名称，用于处理拖动开始事件的函数名称，以及用于处理放下事件的函数名称。

拖动开始的函数接受三个参数：起始 *x* 位置、起始 *y* 位置，以及触发开始事件的 DOM 事件对象。

放下事件接受一个参数：放下时的事件对象。

移动事件处理函数接受 5 个参数：

- `dx`，与起始点的 *x* 方向上的距离
- `dy`，与起始点的 *y* 方向上的距离
- `x`，鼠标的 *x* 位置
- `y`，鼠标的 *y* 位置
- `event`，鼠标移动的 DOM 事件对象

你需要记住圆的起始位置为：

```
var startX = 100;
var startY = 100;
```

下面是你需要添加到 `init()` 中的代码，用于处理圆的拖拽事件：

```
Snap('#circle').drag(dragMove, dragStart, dragEnd);
```

下面是这些函数（以逻辑顺序开始、移动、结束）：

```
function dragStart(x, y, evt) {
  // 找出圆的当前位置
  startX = parseInt(Snap("#circle").attr("cx"), 10);
  startY = parseInt(Snap("#circle").attr("cy"), 10);
}

function dragMove(dx, dy, x, y, evt) {
  Snap("#circle").attr({cx: (startX + dx), cy: (startY + dy)});
}
function dragEnd(evt) {
  // 不需要做什么
}
```

结果见图 14-3，为了节省版面做了一定修改。



图 14-3：拖拽圆的截屏

你可以在线上版本中亲自进行点击和拖动：

http://oreillymedia.github.io/svg-essentials-examples/ch14/snap_events.html

这些例子只是你能用 Snap 库做的事情中的很小一部分。更复杂的例子可以访问它的官网查看 demo。D3、Snap 和 Raphaël 并不是仅有的选择，但它们都有一个共同点：使得使用 JavaScript 动态创建和维护 SVG 更容易。

第 15 章

生成SVG

前面几章已经介绍了 SVG 的主要特性。所有的示例都比较简单，都是使用非常原始的文本编辑器编写的。对于比较复杂的图形，极少有人从头开始编写 SVG。事实上几乎没有人手工编写 SVG。图形设计师通常会使用某些图形软件来生成 SVG，而程序员则使用脚本将原始数据转换为 SVG。

如果你需要处理的图形已经是图形处理程序输出的 SVG 格式，那么几乎就没什么额外的工作了。如果你读过图形软件生成的 SVG 代码，会发现非常难读。一些程序并不会优化 SVG 文件，比如使用分组（<g> 元素）或者优化路径。如果你使用这些程序，那么轻松生成 SVG 的代价就是失去完全手工编写时全盘掌控的能力。好在如果你能理解其中的原理，就仍然可以编写出能操作图形程序输出 SVG 代码的程序。

从数据文件生成 SVG 是一个很麻烦的主题。其中的各种可能性取决于你手上的数据类型和你能使用的编程语言类型。

生成 SVG 的一种方法是使用第 14 章介绍的方法生成 SVG 文档对象模型（DOM）。我们在 14.5 节中提到过 D3.js，它是专门为在 Web 浏览器中使用数据文件动态构建 SVG 图形而设计的。Scott Murray 的 *Interactive Data Visualization for the Web*（O'Reilly）很适合入门者阅读。这个库的原作者 Mike Bostock 和其他在项目 wiki 页（https://github.com/mbostock/d3/wiki/Tutorials）中列出来的人，也有很多非常不错的教程。

另一种动态生成 SVG 的方法是将一些 SVG 代码片段连在一起，然后使用你喜爱的编程语言将它们写成文件。本章第一部分会概述如何使用自己编写的程序将非 XML 格式的地理信息文件转换为 SVG 文件。

如果你的数据已经是 XML 格式，你可能只需要将相关的数据提取出来，然后塞进 SVG 文件框架中即可。这种情况下，你可以使用实现了 XSLT（Extensible Stylesheet Language Transformations，可扩展样式表转换语言）的工具。XSLT 是一种使用 XML 语法来定义如何将一个 XML 文件转换为另一个 XML 文件的方法。本章第二部分展示了如何使用 XSLT 将 XML 格式的航空天气报告转换为 SVG。

15.1　将自定义数据转换为SVG

如果说有谁的生活一直在和图形打交道的话，那一定是地图绘制员。他们通常会希望得到 XML 标记数据，而 SVG 是将这些数据变为可移植格式的一种非常好的方式。然而在现阶段，仍然有非常多的数据是以自定义格式或者专有格式存储的。

其中一种专有格式是由美国环境系统研究所公司（Environmental Systems Research Institute）开发的，被用于 ArcInfo GIS（Geographic Information System，地理信息系统）。这个系统创建的数据可以以 ASCII 格式导出。导出的文件包含一系列描述多边形的数据，并以一行只有 `END` 的文本结束。每个多边形的第一行由一个整形数字标识（ID）和一个顶点的 *x*/*y* 坐标组成。接下来的每一行代表多边形的一个顶点。含 `END` 的行表示多边形描述结束。下面是一个示例文件：

```
         1      -0.122432044171565E+03       0.378635608621089E+02
      -0.122418712172884E+03       0.378527169597E+02
      -0.122434402770255E+03       0.378524342437443E+02
      -0.122443301934511E+03       0.378554484803880E+02
      -0.122446316168374E+03       0.378610463416856E+02
      -0.122438565286068E+03       0.378683666259093E+02
      -0.122418712172884E+03       0.378527169591107E+02
END
         2      -122.36                      37.82
      -122.378                     37.826
      -122.377                     37.831
      -122.370                     37.832
      -122.378                     378.826
END
END
```

要将这样的文件转换为 SVG，只需要简单地将坐标插入到 `<polygon>` 元素的 `points` 属性中即可。唯一需要注意的是，ARC/INFO 以笛卡儿坐标系保存数据，所以我们需要将 *y* 坐标翻转一下。我们即将讲到的程序接受两个参数：输入文件名和输出 SVG 图形的宽度（像素）。

除了这些参数外，我们还需要一些额外的全局变量来处理数据。

- `lineBuffer` 数组，用于保存每一行按空格分隔后的单词或者数字；
- `singlePolygon` 当前多边形的坐标数组；

- polygonList 所有多边形的 points 字符串数组；
- minX/minY/maxX/maxY 当前已经读取过的坐标中的极值。最小值在初始化的时候应该取正无穷（或者当前编程语言中允许的最大值），最大值在初始化的时候应该取负无穷。这样，任何数跟它们比较的时候都能变成新的最大 / 最小值。

下面的算法假设你有办法以行为单位读取输入文件，并将结果放到输出流或者文件中。具体的做法取决于编程语言，但基本上每种编程语言都可以做到这两件事。

(1) 创建一个程序，一次从输入文件中取一个口令[1]。

```
function get_token()
{
  if ( lineBuffer is empty) // 没有数据了？
  {
    从输入文件中读取下一行;
    去除首尾空白;
    从空白处分割得到口令数组;
    放入lineBuffer;
  }
  从lineBuffer中移除第一个,并返回它
}
```

(2) 主程序（在验证和存储输入参数并初始化其他变量之后）使用嵌套循环处理数据文件。外层循环处理每个多边形的开始和结束，内层循环处理坐标对。每个多边形都以一个索引数字开头，以 END 结尾。整个文件也以 END 结尾。读取索引数字，将 singlePolygon 初始化为一个坐标数组，然后读取坐标，直到遇到 END，将坐标对放入坐标数组 polygonList。重复这个过程，直到下一个口令也是 END。

```
open input-file;

while((polygonNumber = get_token()) is not "END")
{
  singlePolygon = 空列表;

  while((xCoord = get_token()) is not "END")
  {
    yCoord = get_token();
    append (xCoord, yCoord) to singlePolygon;

    // 记录最小/最大坐标值
    minX = min(xCoord, minX);
    maxX = max(xCoord, maxX);
    minY = min(yCoord, minY);
    maxY = max(yCoord, maxX);
  }
  append signlePolygon to polygonList;
}
```

注 1：即上文说的以空白分隔的单词或者数字。——译者注

```
close input-file;
```

（3）处理完输入文件后，polygonList 是一个含有坐标对数组的数组，minX/minY/maxX/maxY 保存着坐标值的最大 / 最小值。但在开始构建 SVG 之前，你需要先算出一个合适的缩放比例，以使数据的 *x* 坐标范围和用户输入的宽度相匹配。通过初始化一些其他的变量来处理输入和输出坐标：

```
deltaX = maxX - minX;
deltaY = maxY - minY;
scale = width / deltaX;
height = deltaY * scale;
height = int(height + 0.5);  // 向上取整
```

（4）SVG 文件本身通过输出标记到文件的方式完成，将从数据中提取的坐标值代入：

```
open output;

print the following to output, replacing variables in {}
with their values:

  <!DOCTYPE svg PUBLIC "-//W3C//DTD SVG1.1//EN"
  "http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

  <svg width=" {width}" height=" {height}"
    viewBox="0 0 {deltaX} {deltaY}">
    xmlns="http://www.w3.org/2000/svg">
  <title>Map constructed from {input-file} </title>
  <g style="fill: none; stroke: black;">
```

（5）处理数据数组，生成多边形对象：

```
polygonNumber = 1;
foreach singlePolygon in polygonList
{
  print '<polyline id="poly {polygonNumber}" points=" ';
  从singlePolygon中移除第一对坐标;

  n = 0; // 坐标索引
  foreach coordinate in singlePolygon
  {
    if (n % 2 == 1) // y坐标
    {
      coordinate = (maxY - coordinate); // 翻转y坐标
    }
    else
    {
      coordinate = (coordinate - minX);
    }
    输出坐标,后面带一个空格;
  }
```

```
    // 为了避免一行太长,每行只输出8个坐标值
      n = (n + 1) % 8;
      if (n == 0) { print a newline }
    }
    print ' /> ';    //关闭多边形
    polygonNumber++;
  }
```

（6）关闭标签，关闭文件：

```
print ' </g>\n</svg>\n';
close output;
```

这段用伪代码写的程序是一个真实的 Perl 程序的简化版。以美国密歇根州的数据和 250 像素的宽度运行这个 Perl 程序会得到图 15-1。之所以选择密歇根州是因为有好几个多边形要画，而且它的轮廓比其他州（比如卡罗拉多州）要有趣一些。数据来自 US Census Bureau Cartographic Boundary Files 网站。



图 15-1：由 ARC/INFO 转换为 SVG

15.2　使用XSLT将XML数据转换为SVG

如果你的数据文件是 XML 格式的，那么 XSLT 可能是将它转为 SVG 的最佳选择。

15.2.1　定义任务

本示例使用 XSLT 从一个 XML 文件中提取信息并放置到 SVG 文件中。源数据是来自 http://w1.weather.gov/xml/current_obs/NNNN.xml 的天气数据，其中的 NNNN 是天气观测站的标识。它的格式是由国家海洋和大气组织定义（http://www.nws.noaa.gov/view/current_observation.xsd）的 OMF（Observation Markup Format，观测标记格式）文档。下面是来自 KSJC 观测站的样本数据（为了避免数据过长，进行了编辑）：

```
<current_observation version="1.0">
  <credit>NOAA's National Weather Service</credit>
  <credit_URL>http://weather.gov/</credit_URL>
  <location>San Jose International Airport, CA</location>
```

```
    <station_id>KSJC</station_id>
    <latitude>37.37</latitude>
    <longitude>-121.93</longitude>
    <observation_time>Last Updated on Jul 15 2014, 7:53 am PDT
      </observation_time>
    <observation_time_rfc822>Tue, 15 Jul 2014 07:53:00 -0700
      </observation_time_rfc822>
    <weather>Overcast</weather>
    <temperature_string>62.0 F (16.7 C)</temperature_string>
    <temp_f>62.0</temp_f>
    <temp_c>16.7</temp_c>
    <wind_string>West at 5.8 MPH (5 KT)</wind_string>
    <wind_dir>West</wind_dir>
    <wind_degrees>290</wind_degrees>
    <wind_mph>5.8</wind_mph>
    <visibility_mi>10.00</visibility_mi>
    <copyright_url>http://weather.gov/disclaimer.html</copyright_url>
  </current_observation>
```

我们需要从报告中提取出工作站、日期和时间、温度、风速风向以及能见度，然后将数据填充到如图 15-2 所示的图表模板中。

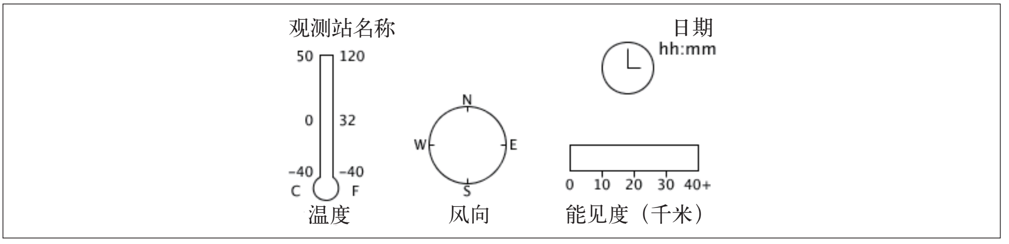


图 15-2：天气图表模板

我们感兴趣的和最终要显示在图表中的数据如下。

- <observation_time_rfc822>

  在图表中，日期和时间会以文本显示，同时时间还会显示在模拟时钟上。在早 6 点到晚 6 点期间，表盘颜色是浅黄色，夜间时间则是浅蓝色。

- <station_id>

  报告数据的观测站标识。图表上会以文本显示。

- <temp_c>

  摄氏温度。温度计上的颜色会根据气温来展示，如果气温高于 0 度，则显示为红色；如果小于等于 0 度，则显示为蓝色。

- <wind_degrees>

  以角度表示的风向。0 表示风向为正北方，270 度表示正西方。会以指南针的指针展示。

- <wind_mph>

  风速，单位为英里 / 小时，会转换为米 / 秒。

- <wind_gust_mph>

  阵风速度，单位也是英里 / 小时，也会被转换为米 / 秒。

- <visibility_mi>

  能见度，单位为英里，将被转换为千米。最后在图表上会显示在水平的条状图表中。超过 40 千米的能见度会被认为是无穷远。

15.2.2　XSLT的工作方式

为了将 OMF 源文件转换为 SVG 格式，我们需要创建一份说明文档来说明 OMF 文件中的哪些元素和属性是我们感兴趣的。说明文档还要详细说明当处理器碰到感兴趣的元素和属性时要怎么生成 SVG 元素。如果你是让另外一个人手工进行这个转换，可以用自然语言编写一份描述。

(1) 以以下内容开始一份 SVG 文档：

```
<!DOCTYPE svg PUBLIC "-//W3C/DTD SVG 1.0//EN",
 "http://www.w3.org/TR/2001/REC-SVG-20010904/DTD/svg10.dtd">
```

(2) 遍历整个源文档，对每个元素都按处理说明进行处理。

(3) 处理 <current_observation> 元素时，在 SVG 文档中添加以下代码，然后按注释中的说明处理子元素：

```
<svg viewBox="0 0 350 200" height="200" width="350">
    <!-- 处理<current_observation>的子元素 -->
</svg>
```

(4) 处理 <station_id> 元素时，在 SVG 文档中添加以下代码，然后将内容填充上：

```
<text font-size="10pt" x="10" y="20">
    <!-- 填充元素内容 -->
</text>
```

(5) 按照“如何画温度计”中的说明处理 <temp_c> 元素。

(6) 按照“如何画风向指南针”中的说明处理 <wind_dir> 元素。

类似地，对其他的元素，也需要指出处理时的说明在哪里，然后在那里给出类似如下的说明。

- 如何画温度计

- 计算高度值，方法为 50 减去温度值。
- 以温度值是否高于 0 来决定使用什么颜色（红色还是蓝色）。
- 将高度和颜色值放入下方斜体显示的地方：

```
<path
  d = "M 25 height 25 90
  A 10 10 0 1 0 35 90
  L 35 height Z"
  style="stroke: none; fill: color;"/>
<path
  d = "M 25 0 25 90 A 10 10 0 1 0 35 90 L 35 0 Z"
  style="stroke: black; fill: none;"/>
```

对“如何画风向指南针”以及其他的元素，也应该有这样详细的说明。

这份说明不是用自然语言编写，也不是由人类来完成转换，而是需要以 XSLT 标记格式来编写。然后可以将 XSLT 和 XML 文件一起交给 XLST 处理器，然后就会生成 SVG 文档中对应的元素，并将对应的值填好。

下面是一份自然语言和 XSLT 的对应表格。

自然语言	XSLT
以指定类型创建输出文档	<xsl:output method="xml" doctype-public="..." doctype-system="...">
输出一个元素	<xsl:template match="element"> <!-- 需要输出的元素 --> </xsl:template>
处理当前元素包含的任何 items 子元素	<xsl:apply-templates select="items">
将 item 的值填充到目标文档	<xsl:value-of select="item">
将 item 的值放到变量 var 中	<xsl:variable name="var"> <!-- 使用 item 的值 --> </xsl:variable>
调用另一个模板 some-name，并传递参数 some-value	<xsl:call-template name="some-name"> <xsl:with-param name="parameter" select="some-value"/> </xsl:call-template>
如果测试结果为真，则使用以下内容	<xsl:if test="some-test"> <!-- 内容 --> </xsl:if>
如果测试结果显真，则使用一段内容，否则使用另外的内容	<xsl:choose> <xsl:when test="some-test"> <!-- 内容 --> </xsl:when> <xsl:otherwise> <!-- 另外的内容 --> </xsl:otherwise> </xsl:choose>

15.2.3　编写XSL样式表

我们会在编写的时候加上一些详细说明，这对于入门已经足够了。XSLT 文件的开头类似这样，在本例中，文件名为 `weather.xsl`：

```
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns="http://www.w3.org/2000/svg">

<xsl:output method="xml" indent="yes"
  doctype-public="-//W3C/DTD SVG 1.0//EN"
  doctype-system="http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd"/>   ❶

<xsl:template match="current_observation">     ❷
<svg width="350" height="200" viewBox="0 0 350 200"
  xmlns="http://www.w3.org/2000/svg">
  <g style="font-family: sans-serif">

  <!-- 处理所有子元素 -->
  <xsl:apply-templates />   ❸
  </g>
</svg>
</xsl:template>
```

❶ `<xsl:output>` 表示输出为 XML 文件，并应该正确缩进。同时它也指定了 `<!DOCTYPE ...>` 说明。

❷ `<xsl:template>` 表示碰到 `<current_observation>` 元素时输出指定内容。内容只会输出一次，因为在源文档中只有一个这样的元素。它创建了最外层的 `<svg>`，并创建了一个后面会用到的 `<g>` 元素。

❸ 在输出 `<svg>` 和 `<g>` 之后，`<xsl:apply-templates>` 表示处理子元素，并应用对应的 `<xsl:template>` 元素。

下面是处理 `station_id` 元素的标记：

```
<xsl:template match="station_id">
 <text font-size="10pt" x="10" y="20">
    <xsl:value-of select="."/>
  </text>
</xsl:template>
```

`<xsl:value-of>` 将指定元素的值插入到当前位置。在这个例子中，`.` 表示“当前元素”。

> 到目前为止，示例中都使用元素名作为 `match` 和 `select` 的值。事实上，你可以使用任何 XPath 表达式作为值。XPath 是一种可以精准选择 XML 中一部分元素的语法。比如在处理 XHTML 文档时，你可选择在 `<tr>` 中的第奇数个且拥有 `title` 属性的 `<td>` 元素。

在一个 <xsl:template> 中输出所有 SVG 相关的内容是可能的，但将内容分成不同的模块则更容易阅读和维护。XSLT 允许你创建一些像函数一样的模板，它们不对应源文档中的任何元素，但你可以使用模板名称显式调用并传入参数。下面是画温度计的代码：

```
<xsl:template match="temp_c">
  <xsl:call-template name="draw-thermometer">
    <xsl:with-param name="t" select="."/>
  </xsl:call-template>
</xsl:template>
```

如果参数值是元素的属性值或者内容，那么设定参数最方便的方法是使用 select，另一种设定值的方式是将内容放在起始标记和结束标记之间。

现在你可以编写模板 draw-thermometer 了。传进来的参数会决定温度计的高度以及使用红色还是蓝色填充。我们分阶段来完成这个示例。首先提取出参数并画出静态部分：

```
<xsl:template name="draw-thermometer">
  <xsl:param name="t" select="0"/>
  <g id="thermometer" transform="translate(10, 40)">
    <path id="thermometer-path" stroke="black" fill="none"
      d= "M 25 0 25 90 A 10 10 0 1 0 35 90 L 35 0 Z"/>

    <g id="thermometer-text" font-size="8pt" font-family="sans-serif">
      <text x="20" y="95" text-anchor="end">-40</text>
      <text x="20" y="55" text-anchor="end">0</text>
      <text x="20" y="5" text-anchor="end">50</text>
      <text x="10" y="110" text-anchor="end">C</text>
      <text x="40" y="95">-40</text>
      <text x="40" y="55">32</text>
      <text x="40" y="5">120</text>
      <text x="50" y="110">F</text>
      <text x="30" y="130" text-anchor="middle">Temp.</text>
  </g>
</g>
</xsl:template>
```

<xsl:param> 元素可以让你指定一个默认值（这里为 0），如果没有参数传入，会使用默认值。

接下来，添加下面的代码，让温度以文字显示出来。如果没有温度值，则显示 N/A。需要注意的是，参数名字为 t，但是如果要引用它的内容，必须写成 $t。将下面这段代码放到 <g id="thermometer-text"> 中：

```
<text x="30" y="145" text-anchor="middle">
  <xsl:choose>
    <xsl:when test="$t != ''">
      <xsl:value-of select="round($t)"/>&#176;C /
      <xsl:value-of select="round($t div 5 * 9 + 32)"/>&#176;F
    </xsl:when>
    <xsl:otherwise>N/A</xsl:otherwise>
  </xsl:choose>
</text>
```

文本内容由 <xsl:choose> 元素根据条件选择，它包含一个或多个 <xsl:when> 元素。test 为真的第一个元素会输出到文档中。如果所有的 test 都不为真，则会使用 <xsl:otherwise> 元素指定的内容。

> 在摄氏转华氏的过程中，使用 div 来做除法，这是因为斜杠 / 已经在 XPath 中被用来区分元素嵌套的层级。

接下来需要填充温度计。只有在 t 不为空字符串时才需要填充。下面的代码使用 <xsl:variable> 来创建一个名为 tint 的变量，并将它的值设为 red 或者 blue，具体取决于温度是否大于摄氏 0 度。XSL 中的变量只能被赋值一次。每次模板被调用的时候，变量会被设为初始值，但是在后续过程中，变量值不能被再次变更。将以下代码放到 thermometer-text 组的结束标记 </g> 之后：

```
<xsl:if test="$t != ''">
  <xsl:variable name="tint">
  <xsl:choose>
    <xsl:when test="$t &gt; 0">red</xsl:when>
    <xsl:otherwise>blue</xsl:otherwise>
  </xsl:choose>
  </xsl:variable>
  <!-- 剩下的代码 -->
</xsl:if>
```

这段代码中给变量赋值的部分再次使用了 <xsl:choose>。test 使用了实体引用 > 表示大于号，以避免 XSLT 处理器在处理时出现问题。如果你需要使用小于符号，需要写成 <。

下面是填充温度计剩下部分的代码：

```
<!-- 填充温度计的过程是画一个实心的矩形
  然后裁剪到需要填充的形状 -->
<xsl:variable name="h">
  <xsl:choose>
    <xsl:when test="$t &lt; -55">
      <xsl:value-of select="105"/>
    </xsl:when>
    <xsl:when test="$t &gt; 50">
      <xsl:value-of select="0"/>
    </xsl:when>
    <xsl:otherwise>
      <xsl:value-of select="50 - $t"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:variable>

<clipPath id="thermoclip">
```

```
    <use xlink:href="#thermometer-path"/>
  </clipPath>
  <path d="M 10 {$h} h40 V 120 h-40 Z"
    fill="{$tint}" clip-path="url(#thermoclip)"/>
```

上面代码中的 <xsl:choose> 有两个 <xsl:when> 子句，它们限制了“水银柱”的高度，以防止温度超过温度计范围时出现异常。<xsl:otherwise> 子句设置了当温度在温度计范围内时的“水银柱”高度。

当我们在输出文档的属性值中使用参数或者变量时，必须使用大括号包裹，正如最后的 <path> 元素。

我们可以看一下到现在为止的转换成果了。在测试之前，你还需要添加一个空模板来处理文本节点。XSLT 处理器会内置一些模板来保证能遍历到源文档中的所有元素和文本。默认的行为是将元素中的文本直接放到目标文档中。在我们的示例中，希望将不需要处理的文本扔掉，所以需要一个空白模板来处理文本节点，保证它们不出现在 SVG 文件中。最后你需要关闭 </xsl:stylesheet> 标记：

```
<xsl:template match="text()"/>

</xsl:stylesheet>
```

调用 XSLT 处理器处理天气数据 XML 文件后，结果见图 15-3。图上显示了气象观测站的名字和温度计。如果你没有独立的 XSLT 处理器，可以在 XML 文件中加入以下几行：

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="weather.xsl"?>
```

然后在浏览器中打开 XML 文件，就能看到转换后的结果。

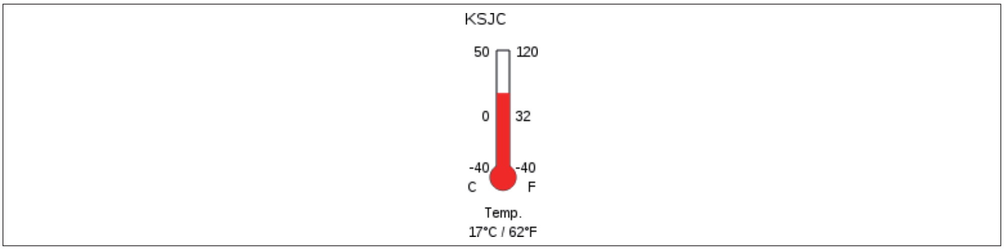


图 15-3：XSL 生成的 SVG 文件

前面你见过 XSLT 可以做一些简单的运算了，其实它还可以做一些字符串操作。下面是显示日期和时间的 XSLT，它使用了 substring 函数来获取需要的信息。

```
<xsl:template match="observation_time_rfc822">
  <xsl:variable name="time" select="."/> ❶
```

```
    <text font-size="10pt" x="345" y="20" text-anchor="end">
      <xsl:value-of select="substring($time, 6, 11)"/> ❷
    </text>

    <xsl:call-template name="draw-time-and-clock"> ❸
      <xsl:with-param name="hour"
        select="number(substring($time, 18, 2))"/>
      <xsl:with-param name="minute"
        select="number(substring($time, 21, 2))"/>
    </xsl:call-template>
  </xsl:template>
```

❶ 为了方便，将字符串存到了一个变量中，避免出现一堆 `<xsl:value-of>`。

❷ `substring()` 函数需要字符串、起始位置索引、截取的字符数量。第一个字符的索引是 1，而不是像很多编程语言一样是 0。

❸ 将小时和分钟的处理工作传给一个模板去做。`number()` 函数将字符串转换成数字类型。

下面是绘制表盘并以文本显示时间的模板（唯一的新东西就是 `format-number()` 函数）：

```
<xsl:template name="draw-time-and-clock">
  <xsl:param name="hour">0</xsl:param>
  <xsl:param name="minute">0</xsl:param>

  <!-- 上午6点到下午6点间,表盘是浅黄色,其他时间是浅蓝色 -->
  <xsl:variable name="tint">
    <xsl:choose>
      <xsl:when test="$hour &gt;= 6 and $hour &lt; 18"
           >#ffffcc</xsl:when>
      <xsl:otherwise>#ccccff</xsl:otherwise>
    </xsl:choose>
  </xsl:variable>

  <!-- 计算时针和分针的角度 -->
  <xsl:variable name="hourAngle"
    select="(30 * ($hour mod 12 + $minute div 60)) - 90"/>
  <xsl:variable name="minuteAngle"
    select="($minute * 6) - 90"/>

  <text font-size="10pt" x="345" y="40" text-anchor="end">
    <xsl:value-of select="format-number($hour,00)"/>  ❶
    <xsl:text>:</xsl:text> ❷
    <xsl:value-of select="format-number($minute,00)"/>
  </text>
  <g id="clock" transform="translate(255, 30)">
    <circle cx="20" cy="20" r="20" fill="{$tint}"
            stroke="black"/>
    <line transform="rotate({$minuteAngle}, 20, 20)"
      x1="20" y1="20" x2="38" y2="20" stroke="black"/>
    <line transform="rotate({$hourAngle}, 20, 20)"
      x1="20" y1="20" x2="33" y2="20" stroke="black"/>
  </g>
</xsl:template>
```

❶ format-number($hour,00) 确定输出有两位数字，如果不足会在前面补 0。

❷ <xsl:text> 元素会将它的内容（必须是纯文本）一字不差地输出到文档中。使用 <xsl:text> 可以避免空格问题。如果不使用它的话，换行符和缩进会进入 SVG 的 <text> 元素中，导致图形中冒号周转产生多余的空格。

下面是绘制风速计的代码：

```
<xsl:template match="wind_degrees">
  <xsl:call-template name="draw-wind">
    <xsl:with-param name="dir" select="number(.)"/>
    <xsl:with-param name="speed"
      select="number(../wind_mph) * 1609.344 div 3600"/> ❶
    <xsl:with-param name="gust"
      select="number(following-sibling::wind_gust_mph) *
        1609.344 div 3600"/>
  </xsl:call-template>
</xsl:template>

<xsl:template name="draw-wind">
  <xsl:param name="dir">0</xsl:param>
  <xsl:param name="speed">0</xsl:param>
  <xsl:param name="gust">0</xsl:param>

  <g id="compass" font-size="8pt" font-family="sans-serif"
    transform="translate(110, 70)">
    <circle cx="40" cy="40" r="30" stroke="black" fill="none"/>
    <!-- 风向标记 -->
    <path stroke="black" fill="none"
      d= "M 40 10 L 40 14
      M 70 40 L 66 40
      M 40 70 L 40 66
      M 10 40 L 14 40"/>
    <xsl:if test="$speed &gt;= 0">
      <path d="M 40 40 h 25"
        fill="none" stroke="black"
        transform="rotate({$dir - 90},40,40)"/> ❷
    </xsl:if>
    <text x="40" y="9" text-anchor="middle">N</text>
    <text x="73" y="44">E</text>
    <text x="40" y="80" text-anchor="middle">S</text>
    <text x="8" y="44" text-anchor="end">W</text>
    <text x="40" y="100" text-anchor="middle">Wind (m/sec)</text>
    <text x="40" y="115" text-anchor="middle"> ❸
      <xsl:choose>
        <xsl:when test="$speed &gt;= 0">
          <xsl:value-of select="format-number($speed, 0.)"/>
        </xsl:when>
        <xsl:otherwise>N/A</xsl:otherwise>
      </xsl:choose>
      <xsl:if test="$gust &gt; 0">
        <xsl:text> - </xsl:text>
        <xsl:value-of select="format-number($gust, 0.)"/>
      </xsl:if>
```

```
    </text>
  </g>
</xsl:template>
```

❶ 这里是一个复杂一点的 XPath 表达式，意思是“该元素的父元素”，所以 ../wind_mph 会找到所有 <wind_degree> 的父元素包含的 <wind_mph> 元素（在这个 XML 文件中，只有一个这样的元素）。获取阵风（如果有的话）的表达式则使用了更冗长的 following-sibling:: 语句。

❷ NOAA 规范中规定正北风是 360 度，0 度代表无风。所以在 SVG 中你需要将角度减去 50 度，因为在 SVG 中 −90 度才代表“北方”。

❸ 将风速（和阵风）以文本方式显示出来的逻辑即使在没有 <wind_mph> 和 <wind_gust_mph> 元素时也能正常工作。当一个不存在的元素的内容被转换为数字时，结果是 NaN（Not a Number，非数字）。与 NaN 进行任何比较都会返回 false。这样，当没有 <wind_mph> 元素时，结果会显示 N/A，如果没有 <wind_gust_mph> 时，阵风面板和数字不会输出。

下面是显示能见度条状图的 XSLT 代码。第一个模板在传入第二个模板时将能见度转换为千米。条状图宽度为 100 像素，所以当能见度大于 40 千米时，宽度为 100，否则会按比例缩小：

```
<xsl:template match="visibility_mi">
  <xsl:call-template name="draw-visibility">
    <xsl:with-param name="v" select="number(.) * 1.609344"/> ❶
  </xsl:call-template>
</xsl:template>

<xsl:template name="draw-visibility">
  <xsl:param name="v">0</xsl:param>
  <g id="visbar" transform="translate(220,110)"
    font-size="8pt" text-anchor="middle">

  <!-- 如果有能见度的值,则填充矩形 -->
  <xsl:if test="$v &gt;= 0">
    <xsl:variable name="width"> ❷
      <xsl:choose>
      <xsl:when test="$v &gt; 40">100</xsl:when>
      <xsl:otherwise>
        <xsl:value-of select="$v * 100.0 div 40.0"/>
      </xsl:otherwise>
      </xsl:choose>
    </xsl:variable>
    <rect style="fill:green; stroke:none;"
      x="0" y="0" width="{$width}" height="20"/>
  </xsl:if>

  <rect x="0" y="0" width="100" height="20"
       style="stroke:black; fill:none"/>
  <path fill="none" stroke="black"
    d="M 25 20 L 25 25 M 50 20 L 50 25 M 75 20 L 75 25"/>
```

```
    <text x="0" y="35">0</text>
    <text x="25" y="35">10</text>
    <text x="50" y="35">20</text>
    <text x="75" y="35">30</text>
    <text x="100" y="35">40+</text>
    <text x="50" y="60">
      Visibility (km)
    </text>
    <text x="50" y="75">
      <xsl:choose>
         <xsl:when test="$v &gt;= 0">
           <xsl:value-of select="format-number($v,'0.###')"/> ❸
        </xsl:when>
        <xsl:otherwise>N/A</xsl:otherwise>
      </xsl:choose>
    </text>
  </g>
  </xsl:template>
```

❶ 第一个模板将能见度转换为千里后传给下一个模板。

❷ 能见度条状图宽度为 100 像素，能见度高于 40 千米时设为 100 像素，低于 40 千米时按比例缩小。

❸ 格式化以后会在小数点后输出 3 位，小数点前会加上 0。

将这些所有的代码放到一起，结果见图 15-4。

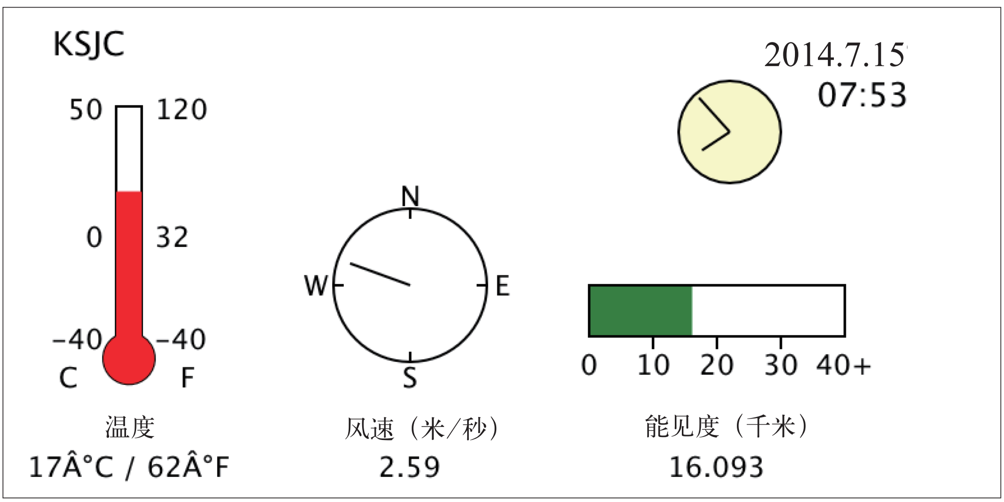


图 15-4：使用 XSLT 生成的 SVG 文件展示了所有数据

这只是 XSLT 所能做的事情中一个小小的示例，想要了解更多的话，可以参考 Doug Tidwell 所著的 *XSLT* 一书（O' Reilly 出版社）。这本神奇的书在第 9 章包含了一个使用 XSLT 从 XML 生成 SVG 文件的示例。如果你经常需要维护 XML 文件，那么建议你购买一本。

附录A

SVG中需要的XML知识

本附录的目标是介绍一下 XML。如果想直接编写 SVG 文档而不是用某些图形工具生成，了解 XML 的相关知识是必要的。

如果你已经熟悉 XML，则无需阅读本附录。如果不熟悉，请继续阅读。本附录中给出的 XML 概述对于处理将要创建的 SVG 文档来说应该绰绰有余。有关 XML 的更多信息，Erik T. Ray 编写的 *Learning XML*（O'Reilly，http://shop.oreilly.com/product/9780596004200.do）以及 Elliotte Rusty Harold 和 W. Scott Means 合著的 *XML in a Nutshel*（O'Reilly，http://shop.oreilly.com/product/9780596007645.do）都是非常不错的参考书。

注意，本附录频繁引用的是正式的 XML 1.0 规范，它所涵盖的主题超过了 SVG 的范围。我们也可以直接参考 Tim Bray 的“带注释的 XML 规范”（http://www.xml.com/axml/testaxml.htm），它提供了一个带有启发性说明的 XML 1.0 规范，也可以参考 Norm Walsh 的 XML 技术介绍（http://www.xml.com/pub/a/98/10/guide0.html）。

你可能注意到这些都不是最近的出版物。不要惊讶，XML 是一个固定的、历史悠久的标准。

A.1　什么是XML

XML，可扩展标记语言，是一种互联网友好的数据和文档格式，由 W3C 发明。标记表示一种在文档内表达文档本身结构的方式。XML 源于一门主要用于发布信息的、名为 SGML（标准通用标记语言）的标记语言，XML 和 HTML 都继承了很多 SGML 的特性。

XML 用来在 Web 上创建机器可读的文档，而 HTML 用来创建人类可读的文档，也就是说，它提供了一种公认的语法，这样底层格式的处理过程就得更为通用，也使得所有用户都能轻易访问这些文档。

然而，和 HTML 不同，XML 的预定义很少。HTML 开发人员都习惯使用尖括号 < > 来表示元素（即语法）以及一系列元素名称（即 head、body 等）。XML 只共享前一个特性，即使用尖括号表示元素。和 HTML 不同，XML 没有预定义元素，仅仅提供了一些规则，从而允许我们编写诸如 HTML 的其他语言[1]。因为 XML 定义很少，对每个人来说都很容易接受它的语法，然后在这个语法的基础上构建应用程序。就像同意使用一组特定的字母和标点符号，但并不意味着一定要在某种语言中才能使用。然而，如果你有 HTML 的开发背景，过渡到 XML 时，需要做好准备自己决定如何命名你的标签。

知道 XML 源于 SGML 应该能帮助我们理解一些 XML 的特性和设计决策。注意，尽管 SGML 本质上是一种以文档为中心的技术，但 XML 的功能还延伸到一些以数据为中心的应用程序，包括 SVG。通常，以数据为中心的应用程序不需要 XML 提供的所有灵活性和表现力，从而限制自己只使用 XML 功能的一个子集。

A.2　XML文档剖析

解释 XML 文档是如何组成的最好的方式就是呈现它。下面的例子展示了一个可能用来描述作者的 XML 文档：

```
<?xml version="1.0" emcoding="us-ascii"?>
<authors>
    <person id="lear">
        <name>Edward Lear</name>
        <nationality>British</nationality>
    </person>
    <person id="asimov">
        <name>Isaas Asimov</name>
        <nationality>American</nationality>
    </person>
    <person id="mysteryperson"/>
</authors>
```

文档的第一行被称作 XML *声明*，用于告诉处理应用我们使用的是哪个版本的 XML（版本标志是强制的）以及这个文档使用的是哪种字符编码。在前面的例子中，文档使用的是 ASCII 编码（关于字符编码的重要性在本章后面讨论）。如果遗漏了 XML 声明，处理程序会对我们的文档作出某些假设。特别是，它期望我们使用 UTF-8 编码——Unicode 字符集编码。然而，最好是尽可能使用 XML 声明，这样既可以避免混淆字符编码，也可以告诉

注 1：澄清一下 XML 和 SGML 的关系：XML 是 SGML 的一个子集。相比之下，HTML 就是一个 SGML 应用。SVG 使用 XML 表达操作，因此它是一个 XML 应用。

处理器我们使用的是哪个版本的 XML。

A.2.1 元素和属性

例子中的第二行以一个*元素*开始，被命名为 authors。这个元素的内容包括 <authors> 的右尖括号 > 和 </authors> 的左尖括 < 之间的所有内容。实际的语法结构 <authors> 和 </authors> 通常分别被称为元素的*起始标签*和*结束标签*。不要将标签和元素混淆！注意，元素可以包含其他元素，以及文本。XML 文档必须包含一个*根元素*，它包含文档中的所有其他内容。根元素的名称定义 XML 文档的类型。

同时包含文本和其他元素的元素被分类为*混合内容*。SVG 的 <text> 元素就是这样的元素。它可以包含文本和 <tspan> 元素。

这个简单的 authors 文档使用名为 <person> 的元素描述作者本身。每个 person 元素都有一个名为 id 的属性。和元素不同，属性只能包含文本型内容。它们的值必须使用引号包裹。单引号（'）和双引号（"）都可以使用，只要我们使用相同引号即可。

XML 文档中，属性经常用于*元数据*（即关于数据的数据）——描述元素内容的属性。这种情况在我们的例子中就是，id 包含一个要描述的 person 的唯一标识符。

就 XML 而言，属性出现在元素起始标签中的顺序是不重要的。例如，这里有两个元素，就 XML 1.0 处理应用程序而言，它们所包含的信息是相同的：

```
<animal name="dog" legs="4"/>
<animal legs="4" name="dog"/>
```

而另一方面，通过 XML 处理器阅读如下两行代码提交给应用程序的信息将会是两个不同的 animal 元素，因为元素的顺序是很重要的：

```
<animal><name>dog</name><legs>4</legs></animal>
<animal><legs>4</legs><name>dog</name></animal>
```

XML 处理一组属性就像把一堆东西装进包里——并没有隐含的顺序问题，而处理元素时就像列表上的项目，是存在顺序问题的。

XML 开发新手经常会问何时使用属性表示信息最好，何时使用元素最好。正如从 authors 例子中可以看到的，如果顺序很重要，那么使用元素是一个不错的选择。总之，并没有一成不变的“最佳实践”告诉我们是选择属性还是元素。

我们的文档中最后描述的 author 并没有什么可用信息。我们只知道这个人的 ID 是 mysteryperson。文档使用了 XML 快捷语法来表示空元素。和下面的形式是等价的：

```
<person id="mysteryperson"></person>
```

A.2.2 命名语法

XML 1.0 对命名元素和属性有一定的规则。特别是：

- 命名区分大小写，比如 <person/> 和 <Person/> 是不一样的；
- 以 xml 开始的命名（包括任意变换的大小写形式）是为 XML 1.0 和它的附属规范预留的；
- 命名必须以字母或者下划线开头，不能是数字，可以包含任意字母、数字、下划线或者句点。[2]

关于命名更准确的描述可以在 XML 1.0 规范的 2.3 节（http://www.w3.org/TR/REC-xml/#sec-common-syn）找到。XML 1.1 的命名规则略有不同，主要是关于 Unicode 字符。SVG 使用的是 XML 1.0 规则。

A.2.3 合法形式

遵从 XML 语法规则的 XML 文档被认为是*合法形式*。从本质上来看，结构的完整性意味着元素必须正确匹配，所有元素都应该闭合。关于结构完整性的正式定义可以在 XML 1.0 规范的 2.1 节（http://www.w3.org/TR/REC-xml/#sec-well-formed）找到。表 A-1 展示了一些不正确的 XML 文档。

表A-1：不正确的XML文档示例

文档	错误原因
<foo> <bar> </foo> </bar>	元素嵌套不正确，因为 foo 闭合在它的子元素 bar 里面了
<foo> <bar> </foo>	bar 元素在它的父元素 foo 闭合之前没有闭合
<foo baz> </foo>	baz 属性没有值。虽然在 HTML 中可以（比如 <table border>），但是在 XML 中不行
<foo baz=23> </foo>	baz 属性的值 23 没有使用引号包裹。和 HTML 不同，XML 中所有的属性值必须用引号包裹

A.2.4 注释

和 HTML 一样，XML 文档中也可以包含*注释*。XML 注释被设计为面向人类可读。对于 HTML 而言，开发人员偶尔会使用特定的注释来添加功能。例如，大多数 Web 服务器都支持的“服务端包含”（Server Side Include，简称 SSI）功能就是以嵌入 HTML 注释的方

注 2：实际上，一个名称还可以包含冒号，但冒号被用来分隔命名空间前缀，并且它也不能随意使用。更多信息请查看 Tim Bray 的“XML 命名空间示例”：http://www.xml.com/pub/a/1999/01/namespaces.html。

式使用的。XML 提供了其他可供应用程序处理的指令[3]，因此注释不应该用于除了说明性文字之外的任何其他目的。

注释的开头用 <!-- 表示，以 --> 结束。任何除 -- 以外的字符序列，都可以出现在注释中。

XML 文档中的注释往往更多的是用于人类理解而不是机器理解。SVG 中的 <desc> 和 <title> 元素可以避免大部分注释需求。

A.2.5 实体引用

编写 SVG 文档时 XML 的另一个特性偶尔会大有用处，就是它的字符转义机制。

由于某些字符在 XML 中具有特殊意义，所以就需要有一种方式来表现它们。例如，某些情况下，< 符号可能真的是要代表小于号，而不是作为元素名的开始符号。显然，仅仅插入这个字符而不做任何转义操作会导致文档不合法，因为处理程序会假定我们是要开启另外一个元素。这一问题的另一个实例是在属性值中需要同时包含双引号和单引号。这里有一个例子，演示了这两个问题：

```
<badDoc>
  <para>
    I'd really like to use the < character
  </para>
  <note title="On the proper 'use' of the " character"/>
</badDoc>
```

XML 中可以使用预定义的实体引用避免这一问题。XML 环境中的单词实体仅仅意味着一个内容单元。术语实体引用意味着，一个象征性的符号表示某个内容单元。XML 预定义了下列实体符号：左尖括号（<），右尖括号（>），撇号（'），双引号（"），和号（&）。

实体符号以 & 开头，紧随其后的是一个名字（真正意义上用的是这个名字，由 XML 1.0 规范定义），最后以分号（;）结尾。表 A-2 展示了可用于 XML 文档中的 5 个预定义实体。

表A-2：XML 1.0预定义的实体引用

原字符	实体引用
<	<
>	>
'	'
"	"
&	&

下面是用实体引用修订有问题的文档之后的结果：

注 3：关于处理指令（PIs）的讨论在本书范围之外。更多关于 PIs 的信息，请参考 XML 1.0 规范的 2.6 节，在 http://www.w3.org/TR/REC-xml#sec-pi。

```
<badDoc>
  <para>
    I'd really like to use the &lt; character
  </para>
  <note title="On the proper &apos;use&apos;
                of the &quot; character"/>
</badDoc>
```

在 SVG 中也能使用这些预定义实体；一般来说，实体为我们创建 XML 文档提供了方便。XML 1.0 还允许我们自定义实体，然后在文档中使用这些实体作为“快捷方式”。XML 1.0 规范的第 4 节（http://www.w3.org/TR/REC-xml/#sec-physical-struct）描述了实体的使用。

A.2.6　字符引用

在 SVG 文档环境中我们还可能找到*字符引用*。字符引用允许我们通过指定 Unicode 字符集中的位置（这个位置也被称为*字符码*）来表示某个字符。表 A-3 中的一些例子演示了它的语法。

表A-3：UTF-8字符引用示例

实际字符	字符引用
1	`1`
A	`A`
Ñ	`Ñ`
®	`®`

注意，这些字符码可以用十进制或者用 x 做前缀的十六进制表示。

A.3　字符编码

字符编码这个主题对开发者来说通常是难以理解的。大多数代码往往都是为某个计算平台编写的，通常情况下运行在一个系统中。尽管互联网正在快速地改变，但我们大多数人从来都没有对国际化进行过深入的思考。

XML 被设计为一种互联网友好的语法，用于信息交换，并且其核心已经国际化了。XML 处理程序最基本的要求之一是，它们要支持 Unicode 标准的字符编码。Unicode 试图将世界上所有的语言包含在一个字符集中。因此，它非常大！

A.3.1　Unicode编码方案

Unicode 3.0 有超过 57 700 个字符码，每个都对应一个字符[4]。如果将每个字符在字符集中的

注 4：你可以在 http://www.unicode.org/charts/ 看到所有的字符。

位置作为编码来表达一个 Unicode 字符（与 ASCII 同样的方式），表达完整的字符范围时每个字符需要 4 个八位字节。[5] 显然，如果文档 100% 使用美式英文编写，将比 ASCII 字符体积大四倍——所有 ASCII 字符都使用 7 位表示。这就要求处理器程序对存储空间和内存作出改变。

幸运的是，有两种 Unicode 编码方案解决了这个问题：UTF-8 和 UTF-16。顾名思义，使用这种编码时应用程序可以一次处理 8 位或 16 位片段的文档。当文档所需的字符码不能通过一个块表示时，会使用一个特定的位来指示字符码需要和接下来的块一起计算。在 UTF-8 中，这通过将第一个八位的最高位设置为 1 来表示。

这种方案意味着 UTF-8 编码在表示拉丁字语言（如英语）时是高效的。在 UTF-8 中，所有 ASCII 字符集都是原生表示的——一个 ASCII 文档和它等价的 UTF-8 定义按字节比较是相同的。

这些知识还能帮助我们调试编码错误。一个常见的错误源于 ASCII 是 UTF-8 的真子集——程序员们习惯了这一事实并生成一个 UTF-8 文档，然后把它们用作 ASCII。当 XML 解析器处理一个包含诸如 Á 这类字符的文档中时会出错。因为在 UTF-8 中这个字符不能只用一个八位字节表示，所以会在文档中输出两个八位字节序列；在非 Unicode 阅读器或者文本编辑器中，它看起来就像一对垃圾字符。

A.3.2 其他字符编码

在计算机历史环境中，Unicode 是一个相对较新的发明。原生操作系统对 Unicode 的支持并不普遍。比如，诸如 Windows 95 和 98 这样的老牌系统就不支持。

XML 1.0 允许文档使用任意在互联网号码分配局（Internet Assigned Numbers Authority，IANA）注册通过的字符集进行编码。欧洲的文档通常使用 ISO 拉丁字符集之一进行编码，比如 ISO-8859-1。日本的文档通常使用 Shift-JIS，而中文文档使用 GB2312 和 Big5。

IANA（http://www.iana.org/assignments/character-sets/character-sets.xhtml）维护了一个完整的注册字符集列表。

XML 1.0 规范并不要求 XML 处理器支持 UTF-8 和 UTF-16 之外的更多编码形式，但是支持其他编码很常见，比如 US-ASCII 和 ISO-8859-1。虽然大多数 SVG 文档当前都是按照 ASCII（或者 UTF-8 的 ASCII 子集）处理的，但是没什么可以阻挡 SVG 文档包含诸其他文字（如韩国文字）。然而，我们可能要探究一下我们使用的计算平台所支持的编码，然后找出可以用来替代的编码。

注 5：一个八位字节就是 8 个二进制数字（也叫比特）组成的串。通常认为字节和八位字节一样，但并非总是如此。

A.4 有效性

除了格式正确之外，XML 1.0 还提供了另一个级别的验证，称作有效性。为了解释为什么有效性很重要，我们来看一个简单的例子。假设我们为朋友的电话号码创造了一个简单的 XML 格式：

```
<phonebook>
  <person>
    <name>Albert Smith</name>
    <number>123-456-7890</number>
  </person>
  <person>
    <name>Bertrand Jones</name>
    <number>456-123-9876</number>
  </person>
</phonebook>
```

基于这个格式，我们还构建了一个程序用于显示和搜索电话号码。这个程序很有用，我们将它共享给朋友。然而，你的朋友并不像我们一样关心细节，然后他试图给这个程序提供一个带有 `<phone>` 元素而不是 `<number>` 元素的电话本文件：

```
<phonebook>
  <person>
    <name>Melanie Green</name>
    <phone>123-456-7893</phone>
  </person>
</phonebook>
```

注意，虽然这个文件的格式是完全正确的，但是它并不符合我们给电话本指定的格式，你会发现需要改变这个程序以应对这种情况。如果你朋友像你一样使用 `number` 表示电话号码，而不是 `phone`，就不会有问题。然而，事实上，第二个文件并不是一个有效的电话本文档。

有效性是一个很有用的普通概念，我们需要一种机器可读的方式告诉它们什么是有效的文档；也就是说，哪些元素和属性必须以什么样的顺序出现。XML 1.0 通过引入文档类型定义（Document Type Definition，简称 DTD）来实现这一点。对 SVG 而言，我们不需要了解太多关于 DTD 的信息。SVG 的确有一个 DTD，它详细准确地阐述了哪些元素和属性组合能生成有效文档。

A.4.1 文档类型定义

DTD 的目的是在特定的文档中告知允许的元素和属性，以及约束它们在该文档类型中显示的顺序。DTD 包含定义元素类型和属性列表的声明。DTD 可能跨多个文件，SVG1.1 规范使用的模块化 DTD 分布在 10 多个文件中。然而，在文件中包含另一个文件的机制——参

数实体——超出了本书的讨论范围。最常见的错误是混淆元素和元素类型。区别在于，元素是 XML 文档结构中的真实实例，而元素类型是该实例元素的类型。

A.4.2 组合在一起

重要的是要知道如何把文档和 DTD 定义连接到一起。我们用文档类型声明 `<!DOCTYPE ...>` 实现这一点，把它插入到 XML 文档的开始部分，跟在 XML 声明后面。下面是一个虚构的例子：

```
<?xml version="1.0" encoding="us-ascii"?>
<!DOCTYPE authors SYSTEM "http://example.com/authors.dtd">
<authors>
  <person id="lear">
    <name>Edward Lear</name>
    <nationality>British</nationality>
  </person>
  <person id="asimov">
    <name>Isaac Asimov</name>
    <nationality>American</nationality>
  </person>
  <person id="mysteryperson"/>
</authors>
```

这个例子假设 DTD 文件已经放在 example.com 服务器上了。注意，文档类型声明指定了文档的根元素，而不是 DTD 本身。我们要使用同一 DTD 定义 `person`、`name` 和 `nationality` 作为有效文档的根元素。某些 DTD，比如用于技术文档的 DocBook DTD[6]，使用这一特性的效果很好，允许我们为多个文档类型提供同一 DTD。

有效的 XML 处理器必须要检查输入文档的 DTD。如果无效，文档要被拒绝。回到前面的电话本例子，如果我们的应用程序验证输入文件符合电话本 DTD，我们就不必调试程序的问题以及纠正朋友的 XML 文件，因为我们的应用程序会拒绝无效的文档。有些读取 SVG 文件的程序有内置的 XML 验证器用于确保输入有效（保持正确性）。A.6 节中讨论了这种 XML 验证器。

A.5 XML命名空间

XML 1.0 允许开发人员创建自己的元素和属性，但是这也可能造成命名冲突。`<title>` 在某个上下文环境中可能意味着一本书的名字，但是在另一个上下文中也可能意味着人名（Ms.、Dr. 等）前缀。XML 规范中的命名空间（http://www.w3.org/TR/REC-xml-names/）提供了一种机制，开发者可以使用统一资源标识符（URIs）标识特定的词汇。

SVG 使用 URI `http://www.w3.org/2000/svg` 作为它的命名空间。URI 只是一个标识符——

注 6：请参考 http://www.docbook.org/。

在浏览器中打开这个地址会跳转到 SVG、XML 1.0 以及 XML 规范中的命名空间。处理程序可以使用命名空间来识别当前位置的词汇代表的确切含义。

SVG 在文档的根元素中应用命名空间：

```
<svg xmlns="http://www.w3.org/2000/svg" width="100" height="100">
....
</svg>
```

xmlns 属性定义命名空间，实际上就是提供 SVG DTD 作为默认值。然而，如果我们不明确使用命名空间，有些浏览器不会渲染 SVG 文档（如果提供了命名空间，就必须是一个准确的值）。命名空间声明会应用给所在元素包含的所有元素，包括容器元素。这意味着名为 svg 的元素也在 http://www.w3.org/2000/svg 这一命名空间中。

SVG 为其内容使用“默认命名空间”，使用 SVG 元素时元素名无需任何前缀。命名空间也可以使用前缀，如下所示：

```
<svgns:svg xmlns:svgns="http://www.w3.org/2000/svg"
    width="100" height="100">
....
</svgns:svg>
```

在这种情况下，命名空间 URI http://www.w3.org/2000/svg 会给所有元素应用 svgns 前缀。SVG 1.0 DTD 不会验证这些文档。

表面上看命名空间非常简单，但是在神秘的 XML 中它是一个众所周知的战场。关于命名空间的更多信息，可以参考 *XML in a Nutshell* 或者 *Learning XML*（都由 O’Reilly 出版）。

A.6　处理XML的工具

许多编程语言都有用于处理 XML 的解析器。大多数都是免费的，绝大多数都是开源的。

A.6.1　选择解析器

XML 解析器通常就是与我们自己的程序配合的代码库。SVG 程序把 XML 移交给解析器，然后返回关于 XML 文档内容的信息。通常，解析器通过事件或者文档对象模型（DOM）做这件事。

对于基于事件的解析，遇到解析事件时解析器会在我们的程序中调用一个函数。解析事件包含查找起始元素、结束元素或者注释。大多数 Java 实现的基于事件的解析器都遵循一个叫作 SAX（http://www.megginson.com/downloads/SAX/）的标准 API，它也对其他语言做了实现，比如 Python 和 Perl。

基于 DOM 的解析器以完全不同的方式工作。它们会消耗整个 XML 输入文档，返回 SVG 软件可以查询和修改的树形数据结构。DOM 是一个有自己的文档（http://www.w3.org/DOM/）的 W3C 标准。

随着 XML 的成熟，混合技术提供了两全其美的方案。如果想找出你所喜欢的编程语言的可用新特性，请留意下面的在线资源：

- XML.com资源指南
  http://www.xml.com/resourceguide/
- 免费XML工具指南
  http://www.garshol.priv.no/download/xmltools/

A.6.2 XSLT处理器

许多 XML 应用都需要将一个 XML 文档转换为另外一个 XML 文档或者是 HTML。W3C 定义了一种叫作 XSLT 的特殊语言来处理这个转换。XSLT 处理器适用于所有的主流编程平台。

XSLT 的工作原理是使用一个样式表，它包含描述如何处理 XML 文档元素的模板。这些模板通常指定 XML 输出特定的元素或者属性。使用一种叫作 XPath 的 W3C 技术不仅可以让我们灵活地做到“对每个 person 元素做某个操作”，还能使用更复杂的指令“对 name 属性为 Fred 的第三个 person 元素做某个操作”。

由于这非常灵活，有些为 XSLT 兴起的应用并不是真正的转换应用，而只是利用了在某些指定的元素模式和顺序上触发操作的能力。XSLT 还允许执行自定义代码，与 XSLT 结合后，XPath 就可以让 XSLT 处理器来驱动一些应用程序，比如索引程序。第 15 章中可以看到 XSLT 的简单介绍。

XSLT 和 XPath 对应的 W3C 规范分别位于 http://w3.org/TR/xslt 和 http://w3.org/TR/xpath。

附录B

样式表介绍

在第 5 章，我们说过 SVG 元素的一些属性可以控制元素的几何性质。比如 `<circle>` 的 `cx` 属性可以控制圆的中间点 *x* 方向的位置。另外还有一些属性可以控制元素的表现，比如 `fill`。而样式表提供了一种方法，可以将表现和几何性质分离，这样如果很多元素都使用同一个样式表的话，只要改动样式表中的一个地方，就可以改变很多不同的 SVG 元素（甚至 SVG 文档）的表现。

B.1　样式的结构

按照规范，一条样式由元素上的一个视觉属性以及为该属性指定的值组成。属性名和值之间使用冒号分隔。例如，要将某元素的轮廓指定为蓝色，那么对应的样式为：

```
stroke: blue
```

要指定多个样式属性的话，需要使用分号将这些属性分隔开。下面的样式指定了轮廓颜色为红色，宽度为 3 像素，填充色为淡蓝色。最后一个属性后面也跟了一个分号，这并不是必需的，但是这样看起来更统一。

```
stroke: red; stroke-width: 3px; fill: #ccccff;
```

B.2　内联样式：style属性

一旦决定为某个元素应用特定的视觉样式，首先要选择需要应用样式的元素。为单一元素指定样式，最简单的方法是将样式写在 `style` 属性中。所以，如果你希望将前面的样式应

用到文档中的一个 <circle> 元素中，可以这样写：

```
<circle cx="50" cy="40" r="12"
    style="stroke: red; stroke-width: 3px; fill; #ccccff;"/>
```

B.3　内嵌样式表

如果你希望将样式应用到单个文档中所有的 <circle> 元素上，那么可以添加一个内嵌样式表。样式表由选择器（应用样式的元素名称）和元素应用的样式组成。其中样式部分被包裹在一对大括号中。下面的样式表会应用到 <circle> 和 <rect> 元素上：

```
<style type="text/css"><![CDATA[
  circle {
    stroke: red; stroke-width: 3px;
    fill: #ccccff;
  }
  rect { fill: gray; stroke: black; }
]]></style>
```

当你在 SVG 文档中添加 <style> 元素的时候，应该将样式内容包裹在 <![CDATA[和]]> 中。这样的写法可以让 XML 解析器知道这里面的内容只是字符内容，在任何情况下都不应该被当成 XML 的结构来解析。

因为样式表被内嵌在文档中，所以它只会应用到单个文档中。如果你希望在很多文档中应用样式，让这些文档中所有的圆和矩形都应用前面的样式，则需要将前面的样式（不含 <style> 和 <!CDATA[）放到一个单独的文件中，比如叫 myStyle.css。然后在各个 SVG 文档中插入下面的代码：

```
<?xml-stylesheet href="myStyle.css" type="text/css"?>
```

接下来，如果要让所有的矩形都填充上淡绿色，并将轮廓改为深绿色，则只需要修改 myStyle.css 即可：

```
rect {fill: #ccffcc; stroke: #006600;}
```

然后重新加载 SVG 文档，就会看到绿色的矩形，而不是灰色的矩形。

B.4　样式类

上面的样式表会影响所有的 <rect> 和 <circle> 元素。假设我们只需要在一部分圆上加上样式，则需要使用样式类。如下方的代码，在 circle 后面加上一个点，再加上一个类名：

```
circle.special {
  stroke: red; stroke-width: 3px;
  fill: #ccccff;
```

```
}
```

如果 SVG 文档中有如下元素，则第一个元素会是默认样式（黑色填充，无轮廓），第二个元素会应用在样式表中定义的样式，因为它的类名与样式表中的类名匹配：

```
<circle cx="40" cy="40" r="20"/>
<circle cx="60" cy="20" r="10" class="special"/>
```

还可以定义一个可以应用于任何元素的通用样式类。假设有很多不同的图形对象都作为警告标识，你可能希望它们的填充色是黄色，并带有红色的轮廓。你可以使用一个只有类名的样式选择器：

```
.warning { fill: yellow; stroke: red; }
```

这样的通用样式类可以应用到任何 SVG 元素上。在下面的例子中，矩形和三角形都会是黄色填充和红色轮廓：

```
<rect class="warning" x="5" y="10" width="20" height="30"/>
<polygon class="warning" points="40 40, 40 60, 60 50"/>
```

`class` 属性可以包含很多类名，使用空白分隔，这些类名中的属性会叠加应用到当前元素上。下面的代码为前面的例子添加了一个新的通用类名 `seeThrough`，效果是半透明，应用到三角形上：

```
<svg width="100" height="100" viewBox="0 0 100 100">
  <style type="text/css"><![CDATA[
    .warning { fill: yellow; stroke: red; }
    .seeThrough { fill-opacity: 0.25; stroke-opacity: 0.5; }
  ]]></style>
  <rect class="warning" x="5" y="10" width="20" height="30"/>
  <polygon class="warning seeThrough" points="40 40, 40 60, 60 50"/>
</svg>
```

B.5 在SVG中使用CSS

问题来了：哪些 SVG 元素的属性也可以在样式表中声明？表 B-1 是一个可以在样式表中使用的属性列表，包括属性名、合法值（默认值以粗体显示）以及可以应用的元素。这个表从 SVG 标准中的属性索引部分（http://www.w3.org/TR/SVG/）修改而来。[1]

`fill` 和 `stroke` 的值类型为 `paint`，也就是以下几种之一：

- none

注 1：W3C 版权所有，2001（麻省理工学院，法国国家信息与自动化研究所，庆应义塾大学）。保留所有权利。http://www.w3.org/Consortium/Legal/。

- currentColor
- 颜色值，详见 4.2.2 节
- 以 url(...) 格式指定的渐变或模式

为了防止加载渐变或者模式时出错，可以指定回退的属性值，只需要用空格分隔即可，且将首先想用的值放前面。

表B-1：SVG的CSS属性表

名称	值	应用元素
alignment-baseline	**auto**|baseline|before-edge|text-before-edge|middle|after-edge|text-after-edge|ideographic|alphabetic|hanging|mathematical	<tspan>、<tref>、<altGlyph>、<textPath>
baseline-shift	**baseline**|sub|super|percentage|legnth	<tspan>、<tref>、<altGlyph>、<textPath> 元素
clip-path	uri	包裹元素和图形元素
clip-rule	nonzero|evenodd|class=noxref	在元素中的图形元素
color	color	用于为 fill、stroke、stop-color、flood-color、lighting-color 提供潜在的非直接值（currentColor）
color-interpolation	auto|**sRGB**|linearRGB	包裹元素、图形元素和
color-interpolation-filters	auto|sRGB|**linearRGB**	滤镜基元
color-profile	**auto**|sRGB|name|uri	引用栅格图像的元素
color-rendering	**auto**|optimizeSpeed|optimizeQuality	包裹元素、图形元素和
cursor	uri|**auto**|crosshair|default|pointer|move|e-resize|ne-resize|nw-resize|n-resize|se-resize|sw-resize|s-resize|w-resize|text|wait|help	包裹元素和图形元素
direction	**ltr**|rtl	<text>、<tspan>、<tref> 和 <textPath>
display	**inline**|block|list-item|run-in|compact|marker|table|inline-table|table-row-group|table-header-group|table-footer-group|table-row|table-column-group|table-column|table-cell|table-caption|none	<svg>、<g>、<switch>、<a>、<foreignObject> 图形元素（包括元素），以及子元素（即 <tspan>、<tref>、<altGlyph>、<textPath>）。除了 none 以外，其他的值都会改变图形元素的显示情况
dominant-baseline	**auto**|use-script|no-change|reset-size|alphabetic|hanging|ideographic|mathematical|central|middle|text-after-edge|text-before-edge|text-top|text-bottom	文本内容元素

（续）

名称	值	应用元素
enable-background	**accumulate**|new [x y width height]	包裹元素
fill	参考表末尾处 paint 的描述，默认值为 **black**	图形和文本内容元素
fill-opacity	不透明度（默认值为 1）	图形和文本内容元素
fill-rule	**nonzero**|evenodd	图形和文本内容元素
filter	*uri*|**none**	包裹元素和图形元素
flood-color	currentColor| 颜色（默认为 black）	<feFlood> 元素
flood-opacity	不透明度（默认为 1）	<feFlood> 元素
font	font-style,font-variant,font-weight,font-size line-height,font-family|caption|icon|menu|message-box|small-caption|status-bar	文本内容元素
font-family	一系列的字体名称或者字体族名称	文本内容元素
font-size	absolute-size|relative-size|length|percentage	文本内容元素
font-size-adjust	数字 |**none**	文本内容元素
font-stretch	**normal**|wider|narrower|ultra-condensed|extra-condensed|condensed|semi-condensed|semi-expanded|expanded|extra-expanded|ultra-expanded	文本内容元素
font-style	**normal**|italic|oblique	文本内容元素
font-variant	**normal** small-caps	文本内容元素
font-weight	**normal**|bold|bolder|lighter|100|200|300|400|500|600|700|800|900	文本内容元素
glyph-orientation-horizontal	角度（默认为 0deg）	文本内容元素
glyph-orientation-vertical	**auto**| 角度	文本内容元素
image-rendering	**auto**|optimizeSpeed|optimizeQuality	图片
kerning	**auto**|length	文本内容元素
letter-spacing	**normal**|length	文本内容元素
lighting-color	currentColor| 颜色（默认值为 white）	<feDiffuseLighting> 和 <feSpecularLighting> 元素
marker、marker-end、marker-mid、marker-start	**none**|*uri*	<path>、<line>、<polyline> 和 <polygon> 元素
mask	*uri*|**none**	包裹元素和图形元素
opacity	不透明度（默认值为 1）	包裹元素和图形元素
overflow	visible|hidden|scroll|auto	建立新的 viewport 的元素、<pattern> 元素和 <marker> 元素
pointer-events	**visiblePainted**|visiblefill|visibleStroke|visible|painted|fill|stroke|all|none	图形元素

（续）

名称	值	应用元素
shape-rendering	**auto**|optimizeSpeed|crispEdges|geometricPrecision	图形
stop-color	currentColor| 颜色值（默认值为 black）	<stop> 元素
stop-opacity	不透明度（默认值为 1）	<stop> 元素
stroke	参考表末尾处 paint 的描述，默认值为 **none**	图形和文本内容元素
stroke-dasharray	**none**|dasharray	图形和文本内容元素
stroke-dashoffset	dashoffset（默认值为 0）	图形和文本内容元素
stroke-linecap	**butt**|round|square	图形和文本内容元素
stroke-linejoin	**miter**|round|bevel	图形和文本内容元素
stroke-miterlimit	*miterlimit*（默认值为 4）	图形和文本内容元素
stroke-opacity	不透明度（默认值为 1）	图形和文本内容元素
stroke-width	*width*（默认值为 1）	图形和文本内容元素
text-anchor	**start**|middle | end	文本内容元素
text-decoration	**none**|underline|overline|line-through|blink	文本内容元素
text-rendering	**auto**|optimizeSpeed|optimizeLegibility|geometricPrecision	元素
unicode-bidi	**normal**|embed|bidi-override	文本内容元素
visibility	**visible**|hidden|collapse	图形元素（包括 <text> 元素）和文本子元素（<tspan>、<tref>、<altGlyph>、<textPath> 和 <a>）
word-spacing	**normal**|length	文本内容元素
writing-mode	**lr-tb**|rl-tb|tb-rl|lr|rl|tb	<text> 元素

附录C

编程概念

很多图形设计师希望使用第 13 章中提到的脚本来操作 SVG。如果他们对编程不熟悉的话，很可能会使用被主流观念叫作“巫毒脚本”（voodoo scripting）的方式来编写代码。这种编程方式就像是背诵一段神秘的咒语，然后希望敌人被吓死。具体的做法就是复制其他人的一段神秘代码到 SVG 文档中，然后希望文档能正常工作。本附录只是做了简单的总结，不要幻想着读完之后就能成为编程大师。我们的目标是介绍一些简单的编程概念，以减少所复制代码的神秘感。我们要讨论的语言是 ECMAScript，它是 JavaScript 的标准化版本。ECMAScript 中使用的很多概念和其他编程语言都是通用的。

C.1　常量

常量是指一个永远不会改变的数字或者字符串，例如 `2`、`2.71828`、`"message"` 和 `'communication'` 等。其中后面两个叫作字符串常量。在 ECMAScript 中，字符串可以使用单引号，也可以使用双引号。这样在编写类似 `"O'Reilly Media"` 或者 `'There is no "there" there.'` 的字符串时就会很方便。

有时你也会看到两种布尔值常量 `true` 和 `false`，用于区分“是 / 否”的场景。

C.2　变量

变量是指在内存中用一段地址保存一个值，这个值可能会变化。你可以把它想象成一个贴有名字的信箱，信箱中保存着写着各种信息的报纸。假设你需要跟踪矩形的当前宽度，并存储一个可变的消息，则在 ECMAScript 中可以这样定义变量：

```
var currentWidth;
var message;
```

可以将它可视化为图 C-1。

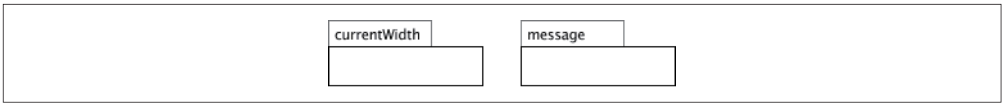


图 C-1：两个空变量

这样定义的变量在“信箱”中没有任何东西，技术术语的描述方式则是这些变量的值未定义。变量名称必须以字母或下划线开始，只能包含字母、数字和下划线。[1] 变量名称是大小写敏感的，所以 `width`、`Width` 和 `WIDTH` 是三个不同的变量。

C.3　赋值和运算

可以通过赋值的方式将一个值放到变量中，具体的语法是变量名后跟一个等号，然后跟上值，比如：

```
currentWidth = 32;
message = "I love SVG.";
```

你可以这样读：“将 `currentWidth` 的值设为 32”和“将 `message` 的值设为 `"I love SVG."`”。事实上，这条语句是从右向左执行的，等号右边的值会赋给等号左边的变量。需要注意的是，ECMAScript 语句以分号结尾。有些情况下分号并不是必需的，但我们也会加上一个，宁可多一个没用的也不要在有用的时候少一个。图 C-2 展示了这两句执行之后的情况。

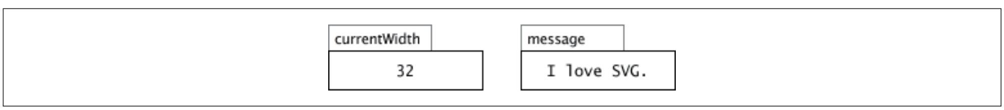


图 C-2：两个被赋值后的变量

事实上，上面的描述并不是特别准确。在等号右边的值其实会先进行运算，然后再赋值给左边的变量。我们可以做一下数学运算：

```
var info;     ❶
info = 7 + 2;     ❷
info = 7 * 2;   ❸
info = info + 1;     ❹
```

注 1：事实上，变量名称等价于 ES5 文档中的标识符 Identifier，它等于标识名称 IdentifierName 除去保留字 ReservedWord 的部分。标识符 Identifier 可使用的字符并非只限字母、数字和下划线。详见 http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf。——译者注

```
info = "door";     ❺
info = info + "bell"    ❻
```

❶ 创建一个名为 info 的空变量。

❷ info 现在的值为 9（7+2）。你还可以使用减号（-）来进行减法运算。

❸ 乘法用星号表示，因为乘号非常容易与字母 x 混淆。info 的值为 14（7 乘以 2）。之前的值 9 会被忽略。你还可以使用斜杠 / 来进行除法运算。

❹ 这在代数中是不合法的，但是在 ECMAScript 中合法。从右边算起：首先获取 info 的当前值 14，然后加 1，右边的运算结果为 15。然后将这个值赋给左边的变量，刚好就是 info。这条语句结束之后，info 的值为 15。

❺ 将字符串 "door" 赋值给 info。在 ECMAScript 中，变量的值可以是任何类型的，并可以随时变更。

❻ 获取 info 的当前值 "door"，然后“加上”（在后面连接）字符串常量 "bell"。结果是单词 "doorbell"，然后通过等号赋值给左边的 info 变量。对字符串来说，加号是唯一可用的运算符号，你不能对它们进行减法、乘法和除法运算。当字符串连接运算和数字四则运算混合在一起的时候，需要特别小心："The answer is " + 2 + 2 的结果为 "The answer is 22"，而 "The answer is " + (2 + 2) 的结果为 "The answer is 4"。

你可以将变量声明和设置初始值放在一起，这叫作初始化变量。你也可以在赋值运算的右侧包含多个运算。下面的代码声明了一个 celsius 空变量，然后声明了一个 fahrenheit 变量，它的值为 212，然后对 fahrenheit 进行运算，赋值给 celsius（华氏温度转摄氏温度）：

```
var celsius;
var fahrenheit = 212;
celsius = ((fahrenheit - 32) / 9) * 5;
```

C.4　数组

数组是指一系列有序数据的集合，使用数字作为索引。前面我们将变量比为邮箱（发送邮件给“张三”），那么数组就是一系列标有数字的邮箱集合（发送邮件给“这堆邮箱中的第 12 个”）。唯一的区别是数组的索引数字从 0 开始，而不是 1。[2] 下面的代码声明了一个表示圆形半径大小的数组（并进行了初始化）。第二条语句将数组的最后一个元素值设为了 9。你可以通过将索引值放到方括号中的方式来访问数组中的元素。图 C-3 展示了代码执行完之后的结果：

```
var radiusSizes = [8.5, 6.4, 12.2, 7];
radiusSizes[3] = 9;
```

注 2：这里不能反过来，因为程序通常会使用数学上的方法来选择指定的条目，从 0 开始会容易很多。

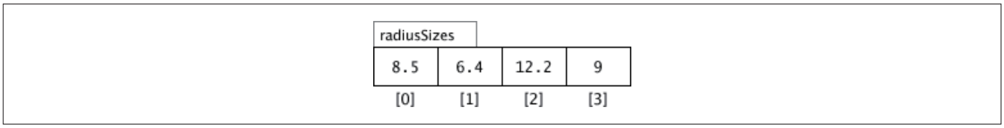


图 C-3：数组示意图

C.5　注释

注释可以让你为自己的程序写上文档，以便其他人更好地了解你写的代码的作用。在 ECMAScript 中，注释的方式有两种。如果你在某行代码中放置两个斜杠（//），则从该位置到行尾的所有字符都被当作注释。如果你希望注释多行，可以使用 /* 和 */ 将它们包裹起来：

```
var interest;   // 这里写一些注释
var rate;       // 十进制,75%表示为0.75

/* 根据上面的变量算出一些值来
   比如$10,000分180个月每月是多少 */
```

C.6　条件语句

通常情况下，你的程序是按照编写时的顺序执行的。有时候会需要根据某个条件执行不同的运算。此时可以使用 if 语句。下面是根据工时计算工资的代码。我们假设所有的变量在前面已经声明过：

```
if (hours <= 40)    ❶
{
  pay = hours * rate;    ❷
}
else     ❸
{
  pay = 40 * rate + (hours - 40) * rate * 1.5;    ❹
}
```

❶ 括号中的表达式叫作条件。它需要的值是“是 / 否”。在这个例子中，则是判断“hours 变量的值是否小于等于 40”。其他比较运算符还包括小于（<）、大于（>）、大于等于（>=）、等于（==）和不等于（!=）。[3] 需要注意的是判断运算符两边是否相等需要使用两个等号！

❷ 如果判断结果为“是”，则程序会继续执行大括号中的代码。

❸ 否则……

注 3：还有 === 和 !== 运算符。当你使用它们时，布尔值和含数字的字符串不会自动转换为数字再比较。

❹ 执行另一对大括号中的代码。大括号的作用是把语句放在一起，表示是一组语句，这和 XML 中使用开始标记、闭合标记来标识内容起止位置类似。

C.7　循环

有时你会想重复一件事一定的次数（比如“从水箱中取水装满 10 个 2L 的容器”）。此时可以使用 for 循环来做。之所以叫循环[4]，是因为如果你把计算机执行你的代码的顺序用箭头画出来，会发现这些逻辑会变成一个个圈。在 ECMAScript 中，上面的装水任务是下面这样的（假设水箱和容器变量已预先定义）。用大括号包裹的循环体中是需要循环执行的动作：

```
var i;              // 计数器
for (i = 0;         // 从0开始计数
     i < 10;        // 计数器最多到10(不包含10)
     i++)           // 每次循环后计数器加1
{
  container[i] = 2;              // 使用数字i装水
  waterTank = waterTank - 2;  // 水箱的水减少2L
}
```

有时也会希望在某些条件为真的时候循环执行一些动作（如“在水箱中有水的情况下一直不停地为 2L 容器装水”）。这种情况可以使用 while 循环：

```
i = 0;                          // 从第一个水箱开始
while ( waterTank > 0)          // 水箱中还有水
{
  container[i] = 2;                // 使用数字i装水
  waterTank = waterTank - 2; // 水箱的水减少2L
  i = i + 1;                 // 下个水箱
}
```

C.8　函数

你可以通过函数来完成一些非常复杂的任务。函数就是一系列 ECMAScript 语句的集合，你可以把它想象成一张标注了食材和做法的食谱，只要按照食谱来做，就能得到指定的菜品。函数以关键字 function 开头，然后接函数名称。为函数命名时一般要求能表达出函数所执行任务的含义，它的具体规则和变量一样。函数名后面接一对括号，里面放函数的参数。参数是指函数在完成任务时需要的信息。就像下面虚构的食谱：

韩国泡菜

每份准备 100g 泡菜，20g 辣椒，50g 蘑菇，混在一起，就好了。

注 4：英文为 loop，有“圈”和“环”的意思。——译者注

做菜之前，你需要先提供一些信息，比如要做多少份。我们的脚本类似这样：

```
function makeKimchiSurprise(numberOfServings)
{
  var kimchi = 100 * numberOfServings;
  var kojujang = 25 * numberOfServings;
  var mushrooms = 50 * numberOfServings;
  var surprise = kimchi + kojujang + mushrooms;
}
```

这只是函数的定义部分。如果不调用它的话，它不会做任何事情。（就像你家里可能有上百份菜谱，只有有人去拿出菜谱来按照上面的说明做的时候才会产出食品。）我们会经常在事件发生时调用某个函数。在下面的例子中，点击蓝色矩形会调用函数。括号中的 5 就是给函数提供的信息，对应函数的 numberOfServings 参数：

```
<rect x="10" y="10" width="100" height="30" style="fill: blue;"
  onclick="makeKimchiSurprise( 5 )" />
```

即使函数不需要参数，也需要有在函数名后面跟上一对括号以调用它。

函数也可以调用其他函数。比如一个计算复利的函数可能会调用另一个函数来确定是否是闰年。计算复利的函数在调用是否闰年的函数时带上一个参数表示年份，是否闰年的函数中的 return 语句会将结果返回给调用者。这样就使得你的程序更加模块化，并使一些通用代码可以被复用。如果用比喻来说，就是 makeHollandaiseSauce()（做辣椒油）函数既可以被 makeEggsBenedict()（做鸡蛋火腿）调用，也可以被 makeChickenFlorentine()（做鸡肉意粉）调用 。

C.9　对象、属性和方法

取一个带开关的电源、一个定时器、一个带弹簧的控制杆、一个矩形金属机箱以及一些线圈。把这些部分组合在一起，你就会得到一个烤面包机。

每一个组成部分都是一个对象。有一些还有独特的属性，比如电源需要的电压是 110 V 或者 220 V，底盘是有颜色的且有一些卡槽，定时器有最大值和最小值等。（控制杆没啥值得关注的特性。）

你的动作也是操作这些对象：推拉控制杆，将面包放入卡槽中，开关电源，旋转定时器等。

我们将这台烤面包机搬到 ECMAScript 中来看看。现在“信箱”中已经可以存入很多份报

纸了，每一份都代表对象不同的属性。简单的数据类型（比如数字和字符串）可以直接写在报纸上，但是更复杂的数据类型就需要有它们自己的“信箱”来存放所有的信息了。所以“一份报纸”代表的是到哪里去找到完整的对象（一个指向数据的指针），这样，多个变量都可以指向同一个对象。一个信箱也可以保存很多份菜谱（函数）供人们选用。当一个变量在另一个“信箱”中时，我们叫它属性。当一个函数在另一个“信箱”中时，我们叫它方法。一个烤面包机的组成如图 C-4。

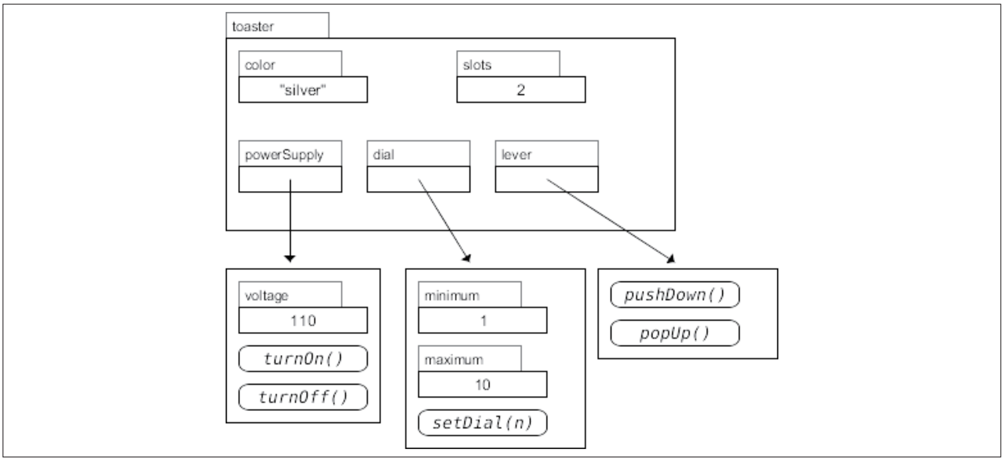


图 C-4：烤面包机对象

这是用来将烤面包机抽象到程序语言的一种非常灵活的方式，但是它也带来了一个问题。为了设置烤面包机的颜色，或者调整它的电压，或者操作控制杆，你不能直接这样：

```
color = "gold";
voltage = 220;
popUp();
```

`color` 属性在 `toaster` 变量中，`voltage` 在 `toaster` 中的 `powerSupply` 中，而操作控制杆则由 `toaster` 的 `lever` 对象调用 `popUp` 方法来完成。因此，你必须这样：

```
toaster.color = "gold";
toaster.powerSupply.voltage = 220;
toaster.lever.popUp();
```

如果你从右往左读，加上“的”，就很容易明白：“将 `toaster` 的 `color` 设置为 `"gold"`”；“将 `toaster` 的 `powerSupply` 的 `voltage` 设置为 `220`”；“调用 `toaster` 的 `lever` 的 `popUp` 方法”。将它想象成小时候听过的故事的升级版就好了：这是一只在追赶杀了吃了在 Jack 建的房子里的麦芽的老鼠的猫的狗。[5]

注 5：由于语法的原因，此句在中文中显得很不通顺，读者理解作者所说的意思即可。——译者注

当我们将烤面包机抽象成一个对象时，就建立了一个“面包机对象模型”。类似地，有一个文档对象模型（DOM）可以让 ECMAScript 获取文档中的属性和调用它的方法。在操作 SVG 文档的过程中，几乎所有的方法都以 set 或 get 开头。比如为了给 id 为 wheel 的 <circle> 元素设置圆角半径，你可以写 svgDocument.getElementById("wheel").setAttribute("r", 3)。在一些情况下，你也可以使用属性。比如，如果你接受到了一个鼠标点击事件，希望获取它的坐标，可以直接用 evt.clientX。

> SVG 文档对象模型其实是 XML 文档对象模型的子集。一旦你学会如何维护 SVG 文档结构，就掌握了如何维护其他的 XML 文档。这样以后再学习 DOM 就会非常快。

C.10　补充说明

我们只是概述了一下“什么是编程”，这可以为你阅读和理解其他人所写的代码提供基础。但如何定义一个任务，如果将它分解为编程中的各个步骤，如何去解决它，则是另一个问题了，这超出了本书的讨论范围。如果你喜欢玩字谜、智力游戏，或者是喜欢解决各种问题，那你很可能会觉得编程很好玩。如果你希望更深入地了解 JavaScript，我们推荐你阅读《JavaScript 权威指南（第 6 版）》。

附录D

矩阵代数

*矩阵代数*是数学的一个分支，它定义了矩阵的操作。所谓*矩阵*就是一系列按行列排列的数字。除了经常用于科学和工程之中，矩阵代数还能用于高效图形计算。本附录的目的是介绍 SVG 中用到的矩阵代数的基本概念。

D.1 矩阵相关术语

通常我们通过行和列的数量来描述矩阵。图 D-1 展示了一个矩阵，包含了两周内每天的温度，被划分成两行七列。这个矩阵也被称作 *2 乘 7 矩阵*。编写时矩阵要包裹在方括号内。

$$\begin{bmatrix} 22.3 & 26 & 27.2 & 24.8 & 28 & 25 & 28.2 \\ 30.3 & 30.4 & 28 & 26.4 & 29.9 & 27.2 & 26 \end{bmatrix}$$

图 D-1：每日气温的 2 乘 7 矩阵

处理矩阵运算时我们还可能遇到其他术语：*方块矩阵*表示具有相同数量的行和列的矩阵。*向量*表示只有一行的矩阵，*列向量*表示只有一列的矩阵。矩阵中独立的数字被称作*元素*，普通数字被称为*标量*。下一次参加聚会讨论时我们就可以谈论这些话题了。

要将矩阵的概念应用到 SVG 中，我们可以使用一个 2 乘 1 矩阵来表示一组 x 和 y 坐标。这并不是我们最终表现坐标的方式，但它是一个很好的起点，因为它很容易理解。图 D-2 就是点 (3, 5) 的表示形式。

$$\begin{bmatrix} 3 \\ 5 \end{bmatrix}$$

图 D-2：使用矩阵表示坐标

D.2　矩阵加法

最简单的矩阵运算就是加法。两个矩阵相加时，将对应的元素相加即可。当然，这就要求我们的矩阵有完全相同的行数和列数。图 D-3 展示了两个矩阵相加，每个都是 3 乘 2 的矩阵。

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 \\ 9 & 10 \\ 11 & 12 \end{bmatrix} = \begin{bmatrix} 1+7 & 2+8 \\ 3+9 & 4+10 \\ 5+11 & 6+12 \end{bmatrix} = \begin{bmatrix} 8 & 10 \\ 12 & 14 \\ 16 & 18 \end{bmatrix}$$

图 D-3：两个 3 乘 2 矩阵相加

我们可以看到，SVG 中的 `translate` 变换可以通过矩阵加法轻松完成。比如，图 D-4 中的矩阵加法可以对任意点 (x, y) 实现 `transform="translate(7, 2)"`。

$$\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} 7 \\ 2 \end{bmatrix} = \begin{bmatrix} x+7 \\ y+2 \end{bmatrix}$$

图 D-4：变换坐标的简单方法

矩阵相加时的顺序无关紧要。从数学上来，说矩阵加法是可交替的（A+B=B+A）。同样可以预想到的是，给定三个矩阵 A、B 和 C，(A+B)+C 和 A+(B+C) 是一样的。另一种运算叫做矩阵减法，就是两个矩阵对应的元素相减。但是和普通的减法一样，矩阵相减是不能交替的。

D.3　矩阵乘法

你可能会想矩阵乘法的工作方式与加法类似，`scale()` 变换就是这样做的。不幸的是，这次简单的方法不奏效了。矩阵乘法比矩阵加法更复杂。在下面的第一个例子中，这种复杂性似乎是不必要的。但是在这个附录的后面，我们会看到矩阵乘法的用处远超它的难度。

要让两个矩阵相乘，第一个矩阵的列数必须与第二个矩阵的行数相等。这样的矩阵是可相乘的。这意味着我们可以用一个 3 乘 5 的矩阵乘以一个 5 乘 4 的矩阵，但不能用 3 乘 5 的矩阵乘以 3 乘 2 的矩阵。图 D-5 中的两个矩阵是可以相乘的。

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}\bullet\begin{bmatrix}7 & 10\\8 & 11\\9 & 12\end{bmatrix}$$

图 D-5：两个矩阵相乘

结果矩阵的行数会和第一个一样，列数会和第二个一样。因此，2 乘 3 的矩阵与 3 乘 2 的矩阵相乘，其结果会是一个 2 乘 2 的矩阵。

结果矩阵行首和列首的条目就是第一个矩阵第一行和第二个矩阵第一列的点积。点积是一种“把行和列中对应条目的乘积加起来”的奇特方式，如图 D-6 所示。

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}\bullet\begin{bmatrix}7 & 10\\8 & 11\\9 & 12\end{bmatrix}=\begin{bmatrix}1\cdot7+2\cdot8+3\cdot9 & -\\- & -\end{bmatrix}=\begin{bmatrix}50 & -\\- & -\end{bmatrix}$$

图 D-6：矩阵乘法中的第一个条目

要得到结果矩阵第二行第一列（左下角）的值，只需要取第一个矩阵第二行和第二个矩阵第一列点积之和即可，如图 D-7 所示。

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}\bullet\begin{bmatrix}7 & 10\\8 & 11\\9 & 12\end{bmatrix}=\begin{bmatrix}1\cdot7+2\cdot8+3\cdot9 & -\\4\cdot7+5\cdot8+6\cdot9 & -\end{bmatrix}=\begin{bmatrix}50 & -\\122 & -\end{bmatrix}$$

图 D-7：矩阵乘法中的第二个条目

生成其余条目计算的结果如图 D-8 所示。

$$\begin{bmatrix}1 & 2 & 3\\4 & 5 & 6\end{bmatrix}\bullet\begin{bmatrix}7 & 10\\8 & 11\\9 & 12\end{bmatrix}=\begin{bmatrix}1\cdot7+2\cdot8+3\cdot9 & 1\cdot10+2\cdot11+3\cdot12\\4\cdot7+5\cdot8+6\cdot9 & 4\cdot10+5\cdot11+6\cdot12\end{bmatrix}=\begin{bmatrix}50 & 68\\122 & 167\end{bmatrix}$$

图 D-8：完成的矩阵乘法

有了这些信息，现在我们可以用矩阵乘法来表示将点 (x, y) 横向放大 3 倍、纵向放大 1.5 倍的运算结果。变换矩阵将是一个两行两列的矩阵，因此它和两行一列的坐标是匹配的，如图 D-9 所示。

$$\begin{bmatrix}3 & 0\\0 & 1.5\end{bmatrix}\bullet\begin{bmatrix}x\\y\end{bmatrix}=\begin{bmatrix}3\cdot x+0\cdot y\\0\cdot x+1.5\cdot y\end{bmatrix}=\begin{bmatrix}3x\\1.5y\end{bmatrix}$$

图 D-9：通过矩阵乘法进行简单缩放

和单一数字乘法不一样，矩阵乘法是不可以交替的。如果有两个矩阵 A 和 B，并且它们不是方块矩阵，那么 A • B 和 B • A 就不会有相同数量的行和列（前提是 A 和 B 在两个方向上都可乘）。即使 A 和 B 都是 3 乘 3 的方形矩阵，也不能保证 A 乘以 B 和 B 乘以 A 的结果相同。实际上，只有在极少数情况下它们的结果才会一样。

还有另外一种有限形式的乘法：一个矩阵乘以一个标量（普通数字），这会用矩阵的每个条目乘以这个标量。正如图 D-10 所示。在缩放中我们并没有提及这一方法，因为这种构造的缩放是均匀的，而 SVG 缩放并不是总是水平方向和垂直方向都一样。

$$5 \bullet \begin{bmatrix} 3 & 2 \\ 5 & 6 \end{bmatrix} = \begin{bmatrix} 15 & 10 \\ 25 & 30 \end{bmatrix}$$

图 D-10：矩阵乘以标量

本质上并没有矩阵除法。但是有一个被称作矩阵求逆的概念，类似于倒数。如果我们让矩阵 A 乘 B 再乘以 B 的逆矩阵，结果就是 A。另一种描述逆矩阵的方式就是 B 的逆矩阵乘以 B 时，会创建一个单位矩阵。单位矩阵就是一个沿着对角线的值为 1、其他位置为 0 的方形矩阵，乘以一个单位矩阵并不会改变原始矩阵。

在 SVG 中，逆矩阵运算通常用于坐标系统之间的转换，因为这里必须计算相反的变换。但是，并非所有的矩阵变换都是可逆的；如果原始转换导致一个或多个维度的矩阵信息丢失，那么后续的乘法就不能够重新创建它。另外，乘以单位矩阵类似于乘以数字 1，乘以不可逆矩阵类似于乘以数字 0。就像试图除以 0，试图对不可逆的矩阵求逆会导致处理器抛出错误。

D.4　如何在SVG变换中使用矩阵代数

我们已经在平移和缩放中用过这种方法了，但是并不理想。例如，如果想先平移一个点，然后缩放它，我们需要先做一个矩阵加法，然后再做一个矩阵乘法。SVG 使用一个巧妙的技巧来表现坐标和矩阵变换，因此我们可以在一个操作中处理缩放和平移。首先，添加第三个值给坐标矩阵，这个值始终等于 1。这就意味着点 (3,5) 将会如图 D-11 所示。

$$\begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix}$$

图 D-11：SVG 坐标

SVG 使用的是设置好额外的 0 和 1 的 3 乘 3 矩阵，用于指定变换。图 D-12 展示了一个水

平平移一个点 tx 距离和垂直平移 ty 距离的矩阵。

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \cdot 3 + 0 \cdot 5 + tx \cdot 1 \\ 0 \cdot 3 + 1 \cdot 5 + ty \cdot 1 \\ 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} 3 + tx \\ 5 + ty \\ 1 \end{bmatrix}$$

图 D-12：SVG 平移

图 D-13 展示了水平方向缩放一个点 sx 倍，垂直方向缩放 sy 倍的变换矩阵。

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 + 0 \cdot 5 + 0 \cdot 1 \\ 0 \cdot 3 + sy \cdot 5 + 0 \cdot 1 \\ 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 \\ sy \cdot 5 \\ 1 \end{bmatrix}$$

图 D-13：SVG 缩放

这就得到了一个一致的表示法；所有的变换，包括旋转和倾斜，都可以用 3 乘 3 的矩阵呈现。此外，由于每个矩阵都是 3 乘 3 形式的，它们都是相互可乘的，所以我们可以把矩阵放在一起建立一系列的变换。比如，要在平移之后进行缩放，我们可以按照这个顺序来添加矩阵（请看图 D-14）。

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix}$$

图 D-14：紧随平移的缩放

和上面的例子一样，这个问题看起来好像也把问题搞复杂了。为了变换点 (3,5)，我们现在需要两个矩阵相乘。要变换另外一个点需要两次或更多的乘法运算。给定一个带有几百个点的 <path> 元素，花费的计算时间相当多。

但是这正是 SVG 聪明的地方：它会将前两个矩阵相乘并存储其结果，如图 D-15 所示。

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} sx & 0 & tx \\ 0 & sy & ty \\ 0 & 0 & 1 \end{bmatrix}$$

图 D-15：乘以平移和缩放矩阵的结果

这个“预先相乘”的矩阵体现了两个变换。通过用这个新的矩阵乘以坐标点的矩阵，平移和缩放就可以在一个矩阵乘法中完成（请看图 D-16）。现在变换一百个点就不需要 200 次乘法，而只需要 100 次了。

$$\begin{bmatrix} sx & 0 & tx \\ 0 & sy & ty \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} 3 \\ 5 \\ 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 + 0 \cdot 5 + tx \cdot 1 \\ 0 \cdot 3 + sy \cdot 5 + ty \cdot 1 \\ 0 \cdot 3 + 0 \cdot 5 + 1 \cdot 1 \end{bmatrix} = \begin{bmatrix} sx \cdot 3 + tx \\ sy \cdot 5 + ty \\ 1 \end{bmatrix}$$

图 D-16：预先相乘平移和缩放矩阵的结果

如果我们要在缩放之后进行旋转，再之后进行一个平移，就要创建三个 3 乘 3 矩阵：一个处理平移，一个处理旋转，一个处理缩放。我们要把所有的这些乘在一起（按照给定的顺序），结果矩阵会体现所有这三个变换需要的计算信息。

正如在 D.3 节中提到的，矩阵乘法是不可交替的。如果我们改变变换矩阵的顺序，就会得到不同的结果。这也是 6.3 节中所描述的变换序列不同导致结果图形有所差异背后的数学原理。

这就是矩阵代数的力量，让我们可以把想要的多个变换信息合并到一个 3 乘 3 的矩阵中，从而大大减少了变换过程中的计算量。图 D-17 中的矩阵用于指定一个旋转 `a` 度，沿 *x* 轴倾斜 `ax` 度以及沿着 *y* 轴倾斜 `ay` 度的矩阵。

$$\begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & \tan(ax) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ \tan(ay) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

图 D-17：旋转、横向倾斜和纵向倾斜变换的矩阵

我们可以使用 `matrix(a,b,c,d,e,f)` 变换来指定填充变换矩阵中的 6 个数字；这个矩阵数字的关系如图 D-18 所示。

$$\begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix}$$

图 D-18：通用的变换矩阵

正如第 11 章中所描述的，SVG 还在滤镜相关的计算中大量使用了矩阵代数。这里，一个像素的红、绿、蓝和阿尔法（不透明度）值被描述为一个 5 行 1 列的矩阵。它还添加了第 5 行，因此一个 5 乘 5 的变换矩阵可以在运算时为这些值加上固定常量以及乘以任何因子。预先乘以坐标变换矩阵的方法也同样适用于像素变换矩阵。

11.3.1 节中介绍的 `<feColorMatrix>` 滤镜允许我们指定所有的 20 个值。这样，标记

```
<feColorMatrix type="matrix"
  values=
```

"a_0 a_1 a_2 a_3 a_4
a_5 a_6 a_7 a_8 a_9
a_{10} a_{11} a_{12} a_{13} a_{14}
a_{15} a_{16} a_{17} a_{18} a_{19}"/>

会被放入如图 D-19 所示的像素变换矩阵中。

$$\begin{bmatrix} a_0 & a_1 & a_2 & a_3 & a_4 \\ a_5 & a_6 & a_7 & a_8 & a_9 \\ a_{10} & a_{11} & a_{12} & a_{13} & a_{14} \\ a_{15} & a_{16} & a_{17} & a_{18} & a_{19} \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

图 D-19：颜色变换矩阵

附录E

创建字体

使用系统内置的字体渲染 SVG 文档能够满足绝大多数需求。但是，有时候我们想要使用自定义字体。从头开始以 SVG 的方式创建一个字体是可行的。简而言之，我们可以用 `<font>` 元素描述来源和字体符号的默认宽度。`<font>` 内是 `<font-face>` 元素，它有大量的属性可以极度详细地描述字体的尺寸。表 E-1 总结了一些比较有用的属性。可以在 SVG 规范（http://www.w3.org/TR/SVG/fonts.html）中详细查看所有信息。

表E-1：font-face属性

font-family	**字体名称列表**
font-style	normal、italic 和 oblique 值之一
font-variant	normal 或者 small-caps
font-weight	normal、bold，或者是以 100 为梯度的从 100 到 900 的数字
font-stretch	说明相对于这个字体系列中其他字体的外观的性质 condensed 或者 expanded。可用于 condensed 或 expanded 的前缀有 ultra-、extra- 和 semi-
font-size	all，适用于大多数可缩放字体，或者是如果字体是按照限定的尺寸、列表长度（比如 18pt）设计的
unicode-range	这个字体所涵盖的 Unicode 字符范围，格式为 *Ustart-end*
units-per-em	em 块的坐标单位。这会给字体建立一个坐标系统。以下属性都按照这个单位测量
cap-height	大写字母高度
x-height	*x* 高度
accent-height	原点到强调字符顶部的距离
ascent	上坡度
descent	下坡度

（续）

font-family	字体名称列表
widths	符号对应每个字符的宽度列表
bbox	字体最大边界框；一个盒子中可以填充的最大字符
underline-position	下划线的理想位置
underline-thickness	下划线的理想厚度

SVG 字体规范旨在允许设计师创建易于访问的商标和图形。搜索引擎和屏幕阅读器会把文本理解为字符序列，但是文本的设计完全可以自定义，并且在每个系统上看起来都一样。编写本书时，这一理想还没有实现，因为两个主流浏览器（IE 和 FireFox）还没有实现 SVG 字体。如果我们创建了一个自定义 SVG 字体，有一些 Web 服务可以把它转换为其他字体格式以便用于浏览器中。

紧随 `<font-face>` 的是 `<glyph>` 元素，用于包含我们想要用于字体的每个符号的路径信息。

虽然从头开始创建字体是可以的，但是工作量很大，并且通常这是一个重复的工作，因为我们需要的符号可能是一个已有的字体。如果对于所需的符号已经有一个 TrueType 字体了，你是幸运的。在 TrueType 中使用二次贝塞尔曲线很容易把字体转换为 SVG 符号。只需要确保在不支持 SVG 字体的浏览器中引入标准的 TrueType 字体作为备选项（作为一个 Web 字体或者引用一个本地字体的名称）即可。

ttf2svg工具

Apache Batik 项目创建了一个工具，可以把 TrueType 字体转换为 SVG。下面的总结改编自 Batik 项目的文档（2013 Apache 软件基金会。保留所有版权）。

如果使用 Batik 二进制发布包，在命令行输入以下命令：

```
java -jar batik-ttf2svg.jar [options]
```

如果使用 Batik 开发者发布包，在命令行输入以下命令：

```
build ttf2svg [options]
```

这两种情况下，选项都是一样的（当在命令行中输入时，这些选项将会在同一行；为了便于阅读，这里我们把它们放在单独的行中）：

```
ttf-path
[-l range-begin]
[-h range-end]
[-ascii]
[id id]
[-o output-path]
[-testcard]
```

选项的含义如下。

- `ttf-path`

  指定包含要转换的字符的 TrueType 字体文件。

- `-l` *`range-begin`*
- `-h` *`range-end`*

  要转换为 SVG 的字符范围的起始值和结束值（ASCII 或者 Unicode 值）。

- `-ascii`

  强制使用 ASCII 字符映射。

- `-id` *`id`*

  指定生成的 `<font>` 元素的 `id` 属性值。

- `-o` *`output-path`*

  指定生成 SVG 字体文件的路径。如果不指定，则输出到 Java 控制台。

- `-testcard`

  用于指定在字体文件中应该插入一系列 `<text>`，用于将字符可视化和测试 SVG 字体中的字符。这提供了一种简单的方式来从视觉上验证生成的 SVG 字体文件。

比如，要转换字符 48 到 57，也就是字符 0 到 9，从 myFont.ttf 到 SVG 等价的 mySVGFont.svg 文件，插入一个测试卡以便轻松将字体可视化，我们可以使用这条命令：

```
java -jar batik-ttf2svg.jar /usr/home/myFont.ttf -l 48 -h 57
   -id MySVGFont -o mySVGFont.svg -testcard
```

在 SVG 文件中嵌入字体之前需要确保你确实有权限进行嵌入操作。TrueType 字体文件包含一个定义字体“嵌入性”的标记，也有些工具可以用来检查这个标记。

附录 F

将圆弧转换为不同的格式

F.1 根据中心和角度转换为SVG

下面的 JavaScript 代码用于将中心和角度格式的圆弧转换为适当的形式，放到 SVG `<path>` 中：

```
/*
 将一个围绕中心的椭圆弧转换为适用于SVG的参数化的椭圆弧

 参数：
   中心x坐标
   中心y坐标
   椭圆x半径
   椭圆y半径
   圆弧开始角度
   圆弧所跨度数
   x轴旋转角度

 返回一个数组，包含：
   弧起点的x坐标
   弧起点的y坐标
   椭圆x半径
   椭圆y半径
   x轴旋转角度
   SVG规范中定义的大弧形标志
   SVG规范中定义的范围标志
   弧末端的x坐标
   弧末端的y坐标
 */
```

```
  function centeredToSVG(cx,cy,rx,ry,theta,delta,phi)
  {
    var endTheta, phiRad;
    var x0, y0, x1, y1, largeArc, sweep;

  /*
    将角度转换为弧度。需要一个单独的变量把phi变为弧度,因为必须保持phi为度数形式用于
返回值
  */
  theta = theta * Math.PI / 180.0;
  endTheta = (theta + delta) * Math.PI / 180.0;
  phiRad = phi * Math.PI / 180.0;

  /*
    找出起点和终点的坐标
  */
  x0 = cx + Math.cos(phiRad) * rx * Math.cos(theta) +
    Math.sin(-phiRad) * ry * Math.sin(theta);

  y0 = cy + Math.sin(phiRad) * rx * Math.cos(theta) +
    Math.cos(phiRad) * ry * Math.sin(theta);

  x1 = cx + Math.cos(phiRad) * rx * Math.cos(endTheta) +
    Math.sin(-phiRad) * ry * Math.sin(endTheta);

  y2 = cy + Math.sin(phiRad) * rx * Math.cos(endTheta) +
    Math.cos(phiRad) * ry * Math.sin(endTheta);

  largeArc = (delta > 180) ? 1 : 0;
  sweep = (delta > 0) ? 1 : 0;

  return [x0, y0, rx, ry, phi, largeArc, sweep, x1, y1];
}
```

F.2 根据SVG转换为中心和角度

下面的代码改编自 Apache Batik 项目，用于将 SVG 风格的弧度转换为中心和角度格式：

```
/*
 转换SVG路径参数形式的椭圆为围绕中心点的圆弧

 参数：
   圆弧起点x坐标
   圆弧起点y坐标
   椭圆x半径
   椭圆y半径
   x轴旋转角度
   SVG规范中定义的最大圆弧标志
   SVG规范中定义的范围标志
   圆弧终点x坐标
   圆弧终点y坐标

 返回一个数组,包含：
```

```
    中点x坐标
    中点y坐标
    椭圆x半径
    椭圆y半径
    圆弧起始角度
    圆弧所跨度数
    x轴旋转角度
  */

function convertArc(x0, y0, rx, ry, xAngle, largeArcFlag,
  sweepFlag, x, y)
{
  // 第1步:计算当前点和终点之间距离的一半
  var dx2 = (x0 - x) / 2.0;
  var dy2 = (y0 - y) / 2.0;

  // 转换角度度数为弧度
  var xAngle = Math.PI * (xAngle % 360.0) / 180.0;
  var cosXAngle = Math.cos(xAngle);
  var sinXAngle = Math.sin(xAngle);

  // 计算x1, y1
  var x1 = (cosXAngle * dx2 + sinXAngle * dy2);
  var y1 = (-sinXAngle * dx2 + cosXAngle * dy2);

  // 保证半径足够大
  rx = Math.abs(rx);
  ry = Math.abs(ry);
  var rxSq = rx * rx;
  var rySq = ry * ry;
  var x1Sq = x1 * x1;
  var y1Sq = y1 * y1;

  var radiiCheck = x1Sq / rxSq + y1Sq / rySq
  if (radiiCheck > 1) {
    rx = Math.sqrt(radiiCheck) * rx;
    ry = Math.sqrt(radiiCheck) * ry;
    rxSq = rx * rx;
    rySq = ry * ry;
  }

  // 第2步:计算(cx1, cy1)
  var sign = (largeArcFlag == sweepFlag) ? -1 : 1;
  var sq = ((rxSq * rySq) - (rxSq * y1Sq) - (rySq * x1Sq)) /
    ((rxSq * y1Sq) + (rySq * x1Sq));
  sq = (sq < 0) ? 0 : sq;
  var coef = (sign * Math.sqrt(sq));
  var cx1 = coef * ((rx * y1) / ry);
  var cy1 = coef * -((ry * x1) / rx);

  // 第3步:根据(cx1, cy1)计算(cx, cy)
  var sx2 = (x0 + x) / 2.0;
  var sy2 = (y0 + y) / 2.0;
  var cx = sx2 + (cosXAngle * cx1 - sinXAngle * cy1);
  var cy = sy2 + (sinXAngle * cx1 + cosXAngle * cy1);
```

```
  // 第4步：计算angleStart和angleExtent
  var ux = (x1 - cx1) / rx;
  var uy = (y1 - cy1) / ry;
  var vx = (-x1 - cx1) / rx;
  var vy = (-y1 - cy1) / ry;
  var p, n;
  // 计算起始角度
  n = Math.sqrt((ux * ux) + (uy * uy));
  p = ux; // (1 * ux) + (0 * uy)
  sign = (uy < 0) ? -1.0 : 1.0;
  var angleStart = 180.0 * (sign * Math.acos(p / n)) / Math.PI;

  // 计算角度范围
  n = Math.sqrt((ux * ux + uy * uy) * (vx * vx + vy * vy));
  p = ux * vx + uy * vy;
  sign = (ux * vy - uy * vx < 0) ? -1.0 : 1.0;
  var angleExtent = 180.0 * (sign * Math.acos(p / n)) / Math.PI;
  if(!sweepFlag && angleExtent > 0)
  {
    angleExtent -= 360.0;
  }
  else if (sweepFlag && angleExtent < 0)
  {
    angleExtent += 360.0;
  }
  angleExtent %= 360;
  angleStart %= 360;

  return( [cx, cy, rx, ry, angleStart, angleExtent, xAngle] );
}
```

作者简介

J. David Eisenberg 是一名程序员和教师，居住在美国加州圣何塞市。David 非常有教学天赋。他开发了 HTML 和 CSS、JavaScript、XML 及 Perl 课程。他在圣何塞的常青谷学院教授计算和信息技术课程。他还开发了在线课程，提供韩语、近代希腊语和俄语的入门教程。David 从 1975 年开始一直在开发教育软件，当时他在伊利诺伊大学的现代外语研究项目组工作，在 PLATO 系统上开发计算机辅助教学。他还是 *Introducing Elixir* 一书的合著者。不编程的时候，David 喜欢数码摄影，照顾一群流浪猫，以及骑自行车。

Amelia Bellamy-Royds 是一名专门从事科学和技术交流的自由撰稿人。她通过参与 Web Platform Docs、Stack Exchange 和 Codepen 等在线社区，帮助推动 Web 标准和设计。她对 SVG 的兴趣源于数据可视化的工作，并建立在她学会编程的基础之上。她拥有生物信息学理学学士学位。在加拿大国会图书馆做政策研究工作时，她发现自己更喜欢讨论科学研究的应用，而不是做实验室工作，于是她开始学习新闻学研究生课程。她目前生活在加拿大阿尔伯塔省埃德蒙顿市。如果不在电脑前，她很可能在料理菜园或者外出享受现场音乐。

封面介绍

本书封面上的动物是一只大眼斑雉。这种雉鸡分布在马来西亚、泰国、苏门答腊岛和婆罗洲岛，生活在热带雨林中。雄性的面部呈蓝色，有黑冠和短头冠，腹部呈斑驳的棕色。翅膀和尾羽上的彩虹色斑点有助于吸引雌性。雌性比雄性体型小，也没有那种华丽的羽毛。

大眼斑雉的翅膀可以持续生长到 6 岁。它的尾羽在所有鸟类中是最长的，达 5.7 英尺。有些文化在他们的头饰中使用这种羽毛。

O’Reilly 封面上的许多动物都濒临灭绝，它们都对这个世界非常重要。要了解更多关于如何提供帮助的信息，请参考 animals.oreilly.com。

封面图片是一个出自 Dover Pictorial Archive 的 19 世纪的雕刻品。

欢迎加入

图灵社区 iTuring.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

关注图灵教育 关注图灵社区

iTuring.cn

在线出版　电子书《码农》杂志　图灵访谈……

QQ联系我们

图灵读者官方群I：218139230
图灵读者官方群II：164939616

微博联系我们

官方账号：@图灵教育 @图灵社区 @图灵新知
市场合作：@图灵袁野
写作本版书：@图灵小花 @图灵张霞
翻译英文书：@朱巍ituring @楼伟珊
翻译日文书或文章：@图灵乐馨
翻译韩文书：@图灵陈曦
电子书合作：@hi_jeanne
图灵访谈/《码农》杂志：@李盼ituring
加入我们：@王子是好人

微信联系我们

图灵教育
turingbooks

图灵访谈
ituring_interview

O'REILLY®

SVG精髓（第2版）

本书详尽介绍了可缩放矢量图形（SVG）技术。SVG是一种标记语言，为大多数矢量绘图程序和交互式Web图形工具所使用。本书将带你详细了解SVG的功能，首先学习简单的SVG应用，如绘制线条，然后逐步探索复杂的特性，比如滤镜、变换、渐变和图案等。

本书第2版扩展了动画、交互式图形以及SVG编程等内容。交互式的在线示例让你很容易在Web浏览器中实验SVG的特性。本书还为经验丰富的设计师准备了6个附录，解释了XML标记和CSS样式等基本概念，因此即使你没有网页设计的经验，也可以开始学习SVG。

通过阅读本书，你将能够：

- **为网页创建高质量、高分辨率的图形；**
- **创建通过搜索引擎或辅助技术易于访问的图表和装饰性标题；**
- **用SVG蒙版、滤镜以及变换给图形、文本和照片添加艺术效果；**
- **用SVG标记动画绘制图形，使用CSS和JavaScript添加交互；**
- **根据现有的矢量数据或XML数据使用编程语言或XSLT创建SVG。**

J. David Eisenberg是一名程序员和教师。他开发了CSS、JavaScript、CGI、XML和Perl等多门编程课程，并在加州圣何塞常青谷学院教授计算机信息技术课程。他还著有*Études for Erlang*、*Let's Read Hiragana*以及本书第1版。

Amelia Bellamy-Royds是一位专门从事科学和技术交流的自由撰稿人。她通过参与Web Platform Docs、Stack Exchange和Codepen等在线社区，帮助推动Web标准和设计。

"早在2002年，我就通过本书的第1版初次了解了SVG，它对我帮助很大。真的很高兴，如今本书针对现代浏览器以及新时代的开发者和设计者进行了更新升级。"

——Doug Schepers
万维网联盟SVG工作组成员

XML/WEB DESIGN

封面设计：Karen Montgomery 张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机/网页制作

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

ISBN 978-7-115-40254-7

9 787115 402547 >

ISBN 978-7-115-40254-7
定价：69.00元

看完了

如果您对本书内容有疑问，可发邮件至 contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱:
ebook@turingbook.com。

在这可以找到我们:

微博 @图灵教育：好书、活动每日播报
微博 @图灵社区：电子书和好文章的消息
微博 @图灵新知：图灵教育的科普小组
微信 图灵访谈：ituring_interview，讲述码农精彩人生
微信 图灵教育：turingbooks